Savonia University Of Applied Science

Emulator of the read-out chain

Emulator of the read-out chain of the TOTEM online software framework, and project management

Risto Kivilahti
1/20/2010

Preface

This report is the result of my master´s thesis carried out at CERN, European organization for nuclear research. The master´s thesis is part of studies at Savonia university of applied science. The thesis consisted from the project management side and from the development side. The project management was done to support the work, to give a deterministic flow. From project management side the thesis concentrates to the SCRUM. From technical side the thesis contains CRC optimization part and emulator implementation part.

Table of Contents

1. Introduction

This chapter provides base information of CERN, Totem experiment hardware and software. The main purpose is to give background knowledge of the environment the thesis is done.

1.1. CERN

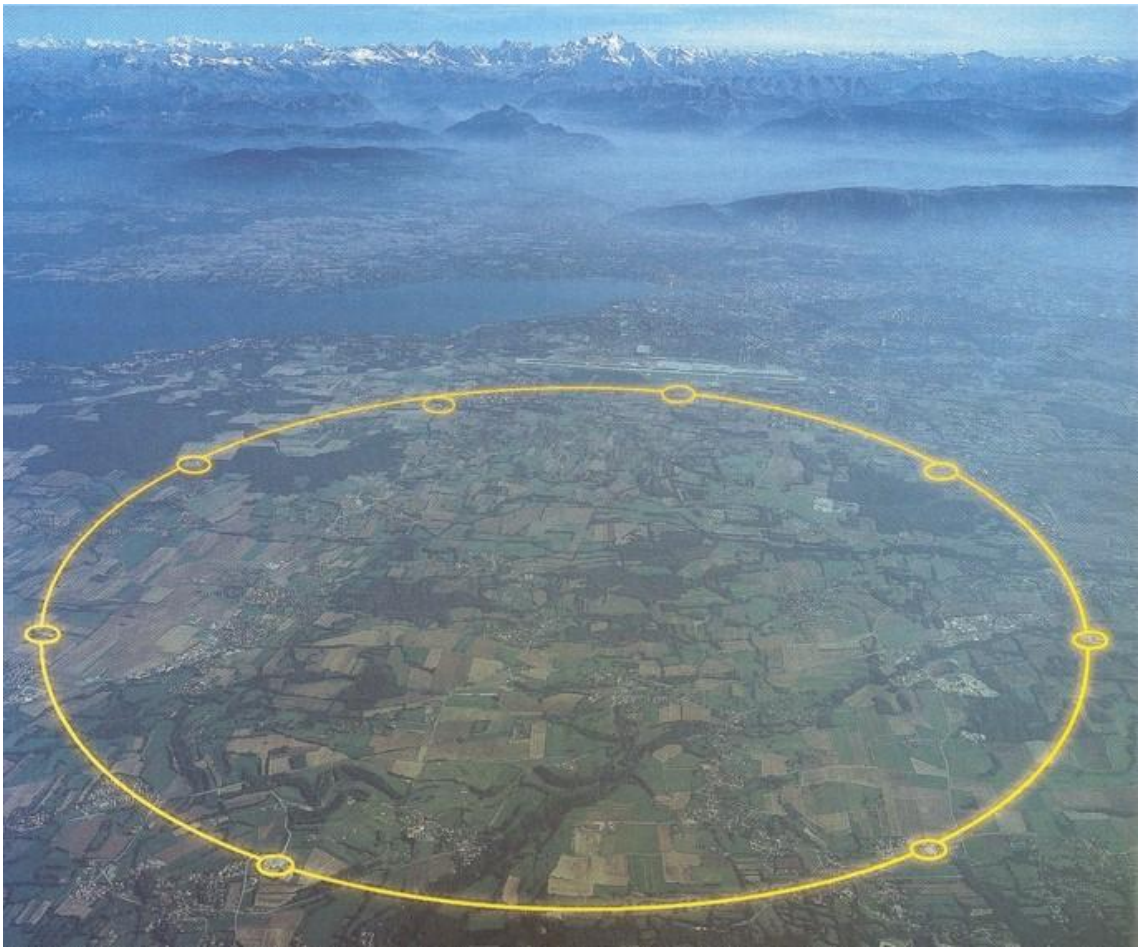
CERN, the European Organization for Nuclear Research, is one of the world's largest and most respected centers for scientific research. Its business is fundamental physics, finding out what the Universe is made of and how it works. At CERN, the world's largest and most complex scientific instruments are used to study the basic constituents of matter — the fundamental particles. By studying what happens when these particles collide, physicists learn about the laws of Nature.

The instruments used at CERN are particle accelerators and detectors. Accelerators boost beams of particles to high energies before they are made to collide with each other or with stationary targets. Detectors observe and record the results of these collisions.

Founded in 1954, the CERN Laboratory sits astride the Franco–Swiss border near Geneva. It was one of Europe's first joint ventures and now has 20 Member States.

1.2. LHC

Large Hardron Collider (LHC) *[1]* is the world's largest particle accelerator, intended to collide opposing particle beams, of either protons or lead nuclei.

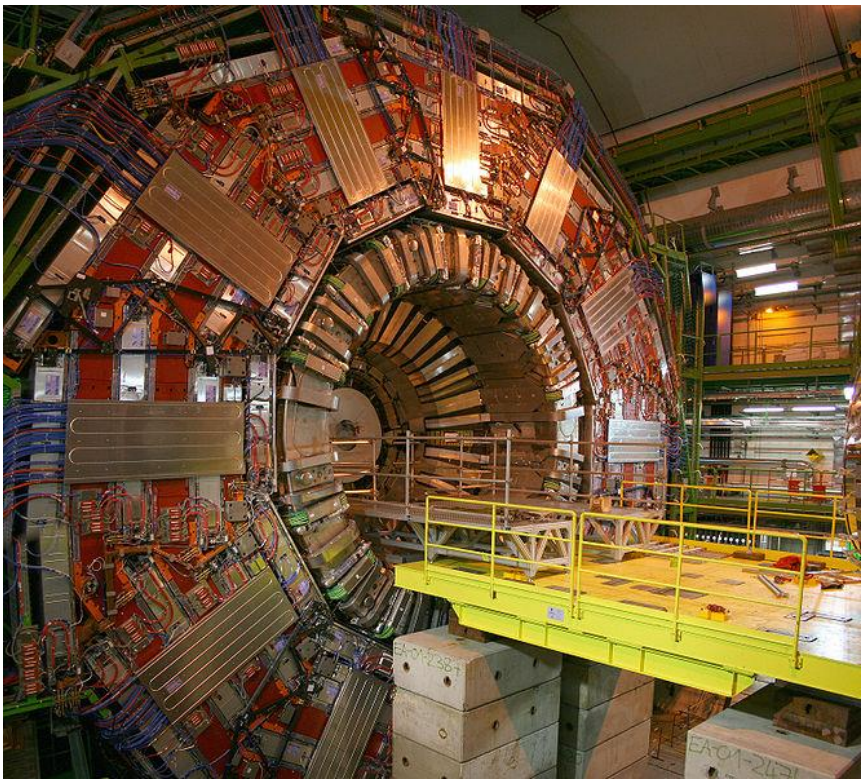


Picture 1.2: The LHC ring, as it goes underground

1.3. CMS

CMS stands for Compact Muon Solenoid: compact because it is “small” for its enormous weight, muon for one of the particles it detects, and solenoid for the coil inside its huge superconducting magnet. It is a high-energy physics experiment in Cessy, France, part of the Large Hadron Collider (LHC) at CERN. CMS is designed to see a wide range of particles and phenomena produced in high-energy collisions in the LHC. Like a cylindrical onion, different layers of detector stop and measure the different particles, and use this key data to build up a picture of events at the heart of the collision.

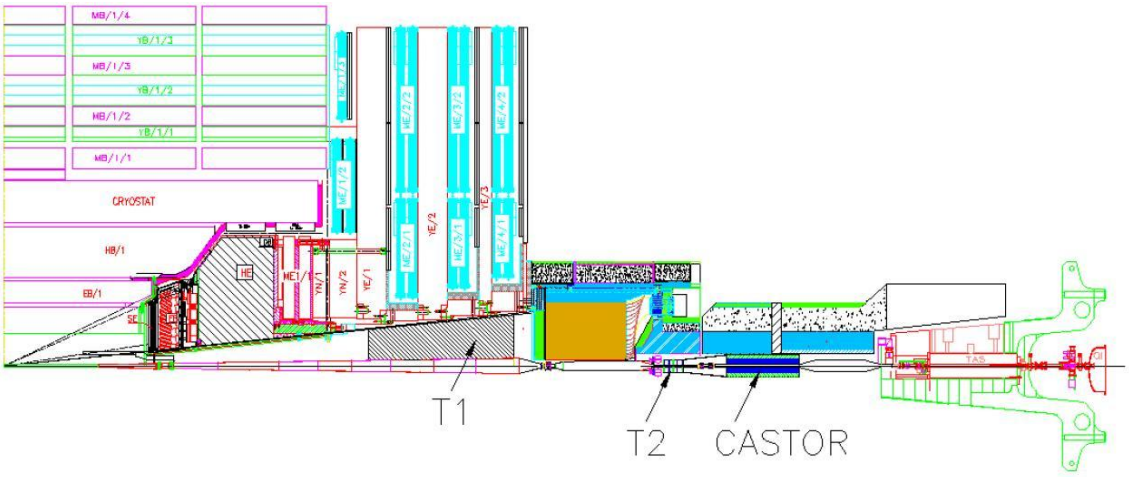
Scientists then use this data to search for new phenomena that will help to answer questions such as: What is the Universe really made of and what forces act within it? And what gives everything substance? CMS will also measure the properties of previously discovered particles with unprecedented precision, and be on the lookout for completely new, unpredicted phenomena. *[2]*



Picture 1.3: The CMS in building phase (2008)

1.4. TOTEM experiment

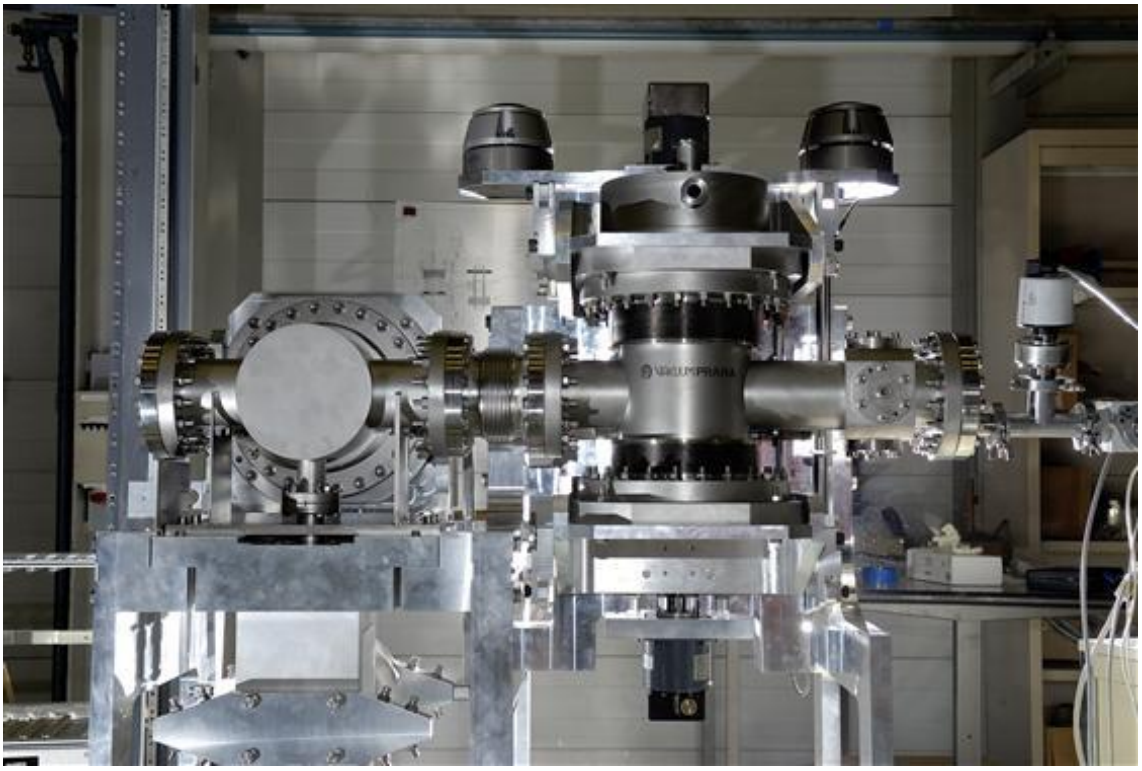
The Total Cross Section, Elastic Scattering and Diffraction Dissociation (TOTEM) *[3]* experiment — small in size compared to the others at the LHC — dedicated to the measurement of the total proton-proton cross-section. TOTEM's physics program aims at a deeper understanding of the proton structure. The experiment is divided to three detectors, Roman Pots, T1 and T2. Each detector is shortly described in the following chapters.



Picture 1.4: The TOTEM forward trackers T1 and T2 embedded in the CMS detector together with the planned CMS forward calorimeter CASTOR.

1.5. Roman Pots

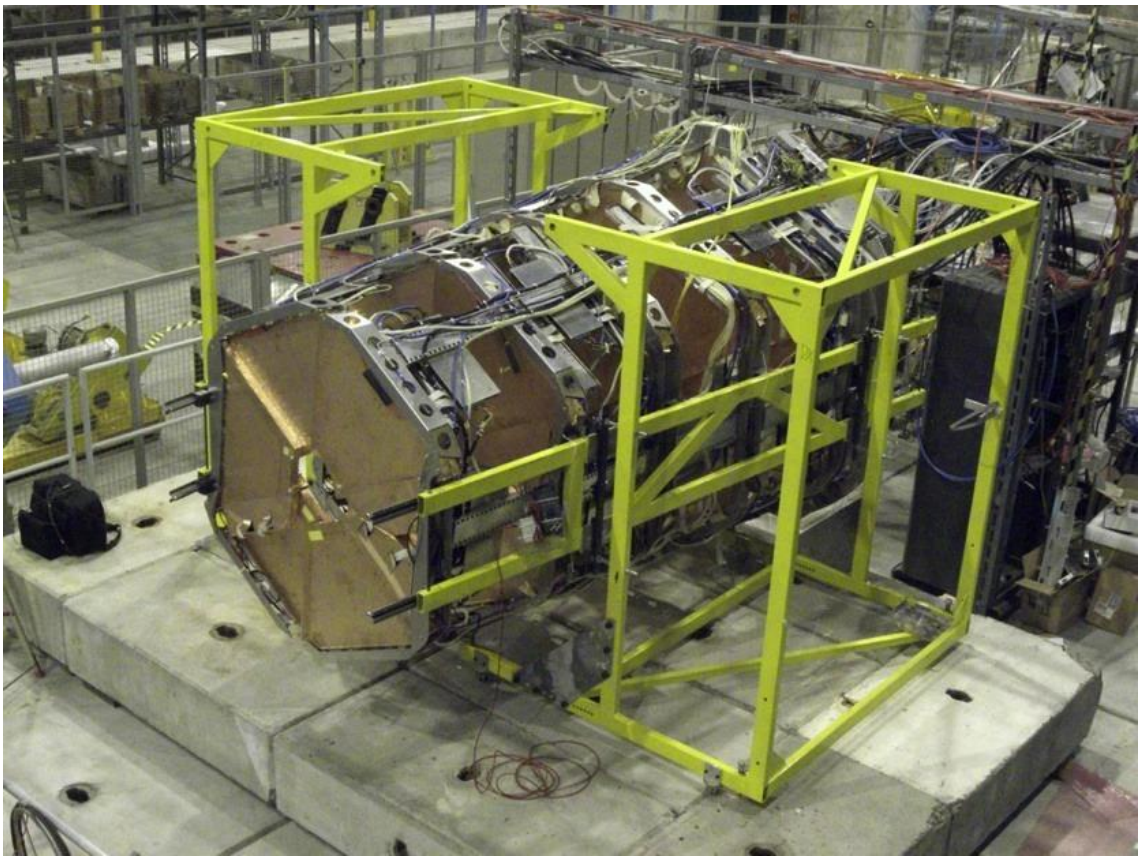
Roman Pots (RP) *[7]* — placed at about 147m and 220m from the interaction point, designed to detect leading protons at merely a few mm from the beam centre. The proton detectors in the Roman Pots are silicon devices designed by TOTEM with the specific objective of reducing the insensitive area at the edge facing the beam to only a few tens of microns.



Picture 1.5: Roman Pots

1.6. T1

The particle telescope closest to the interaction point (T1, placed at 9 m) consists of Cathode Strip Chambers (CSC) *[4].*



Picture 1.6: T1

1.7. T2

The T2 particle telescopes located at 13.5m on both sides of IP5 are detecting charged particles. The gaseous electron multipliers (GEM) *[5]* were selected for detectors of the T2 telescope. The T2 detector consists of 40 GEM detectors arranged in four quarters, each having ten detectors. Each GEM detector consists from so called pads and strips. The pads and strips form a net of wires, the pads creating vertical wires, and the strips creating horizontal wires. This way the coordinates of the particle hit location is defined. Each detector is read out by 17 VFAT microchips, 13 for the pads (120 pads per VFAT) and 4 for the strips (128 strips per VFAT).

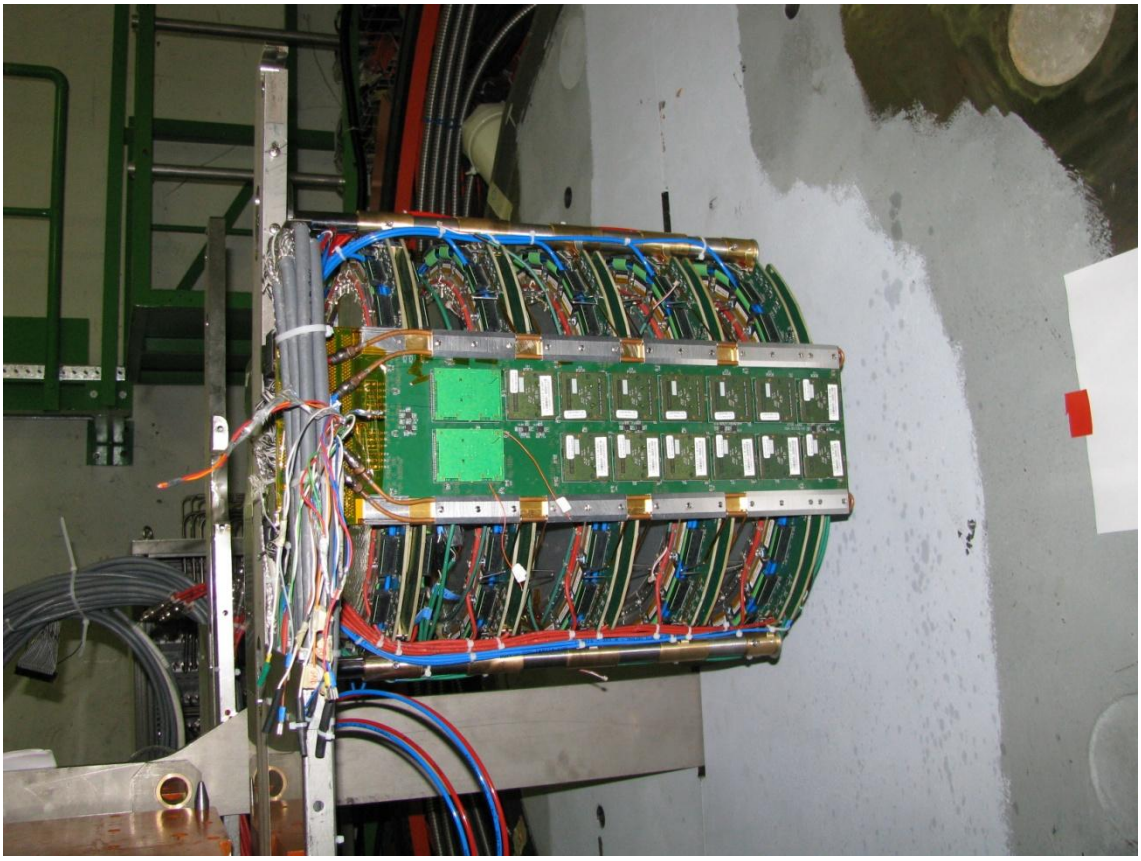


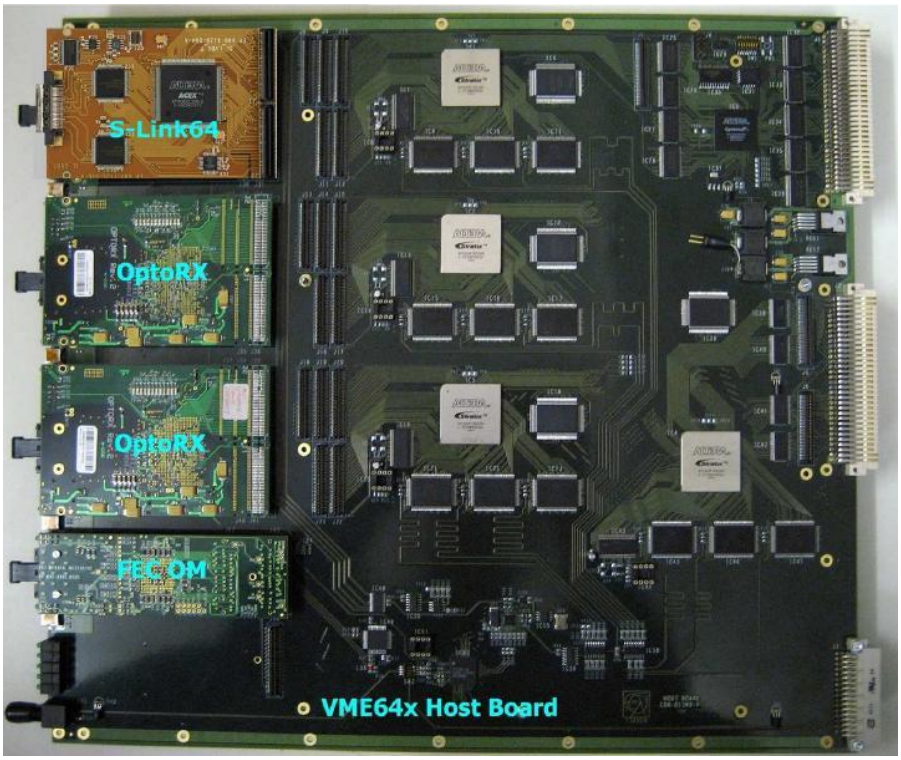
Figure 1.2: One quarter of T2 detector installed.

1.8. Read-out chain hardware

The read-out of all TOTEM subsystems is based on the custom-developed digital VFAT chip *[6]* with trigger capability. The data acquisition (DAQ) system is designed to be compatible with the CMS DAQ to make common data taking possible at a later stage.

The VFAT is the chip which reads the information directly from the sensor. Data packets are transmitted from the VFAT outputs at a bit rate of 40 Mb/s. The data is serialized and transmitted to the counting room via optical links. All VFATs operate synchronously. Once in the counting room the optical fibers are connected to the VME64x Host boards. The incoming optical fibers are connected to optical receiver modules called optoRx-12. From the VME card the data is read by a PC cluster.

The host board is also called as a TOTFED. TOTFED stands for TOTEM Front End Driver.



Picture 1.3: VME64x Host Board

1.9. PC Data Acquisition software

The PC cluster is built on Linux. The data acquisition software is standing on the XDAQ toolkit, which is middleware for distributed data acquisition systems. The XDAQ is under the BSD license. The platform has three components:

- Core Tools contains all required packages to start working with XDAQ.

- Power Pack contains packages that make it easier to develop distributed DAQ systems.

- Work Suite contains DAQ tool programs such as an event builder and readout components.

2. Development tools

Several tools were used in the development process and in this chapter we shortly introduce the tools used.

2.1. Trac

The following is Trac authors comment of what is Trac. *[1]*

Trac is an enhanced wiki and issue tracking system for software development projects. Trac uses a minimalistic approach to web-based software project management. Our mission is to help developers write great software while staying out of the way. Trac should impose as little as possible on a team's established development process and policies.

It provides an interface to Subversion (or other version control system), an integrated Wiki and convenient reporting facilities.

Trac allows wiki markup in issue descriptions and commit messages, creating links and seamless references between bugs, tasks, change sets, files and wiki pages. A timeline shows all current and past project events in order, making the acquisition of an overview of the project and tracking progress very easy. The roadmap shows the road ahead, listing the upcoming milestones.

2.2. Agilo

Agilo is a free plugin for Trac. *[2]* It modifies Trac for easier handling of Scrum like software development processes. This includes things like sprints and user stories. Agilo is not mature software, but it is easily enough for this kind of thesis project.

2.3. gdb

The GNU Debugger, gdb is a debugging tool used from shell. *[3]*

2.4. valgrind

Valgrind is a software analysis tool. [*4]* In the thesis we use valgrind to check memory leaks. Memory tests are done by creating testers for the classes and running those under Valgrind.

2.5. Make

Make belongs to GNU´s not unix autotools family, or more commonly just GNU. *[5]* Autotools have many small applications to make developers life easier. We only use Make because we produce code in own project which will be later integrated to a bigger project. Make is sufficient to

check that all compiles, including the testers. The following listing is official statement of the capabilities of GNU Make.

- Make enables the end user to build and install your package without knowing the details of how that is done -- because these details are recorded in the makefile that you supply.

- Make figures out automatically which files it needs to update, based on which source files have changed. It also automatically determines the proper order for updating files, in case one non-source file depends on another non-source file.

  As a result, if you change a few source files and then run Make, it does not need to recompile all your program. It updates only those non-source files that depend directly or indirectly on the source files that you changed.

- Make is not limited to any particular language. For each non-source file in the program, the makefile specifies the shell commands to compute it. These shell commands can run a compiler to produce an object file, the linker to produce an executable, `ar` to update a library, or TeX or Makeinfo to format documentation.

- Make is not limited to building a package. You can also use Make to control installing or deinstalling a package, generate tags tables for it, or anything else you want to do often enough to make it worthwhile writing down how to do it.

2.6. Editors

Multiple different code editors were used but the main tools were gedit *[6]* and vim *[7]*. These are simple text editors with syntax highlighting property. No integrated development environment (IDE) was used. gedit is a graphical editor provided by Gnome project, and vim is a text based editor used from shell.

3. Development process

Project management keeps track of progress and creates discipline and commitment to complete on time. In this thesis the same person is the worker as well as the manager, but it does not make it less valuable to manage.

The chapter is divided in two sections. The first part contains introduction to the SCRUM. The second part contains an introduction to the development process used in this thesis.

3.1. Scrum

Scrum is a name of a development process. The main goal is to quarantine something deliverable on time and in budget. The other processes try to do the same, but it may not be the main focus to get something delivered. As an example another process may prioritize quality of the design.

The scrum belongs in the agile development process family. In similar way the scrum is iterative meaning simply that the work is done in short iterations. In iteration the currently highest priority features are done. After iteration the product may not be finished, but it should be deliverable. Iteration contains time for requirements, design and implementation. In this way you do not need all the requirements immediately, and you do not have to design the whole software beforehand. In this way the process provides agility to the software development

This thesis does not go deep into the scrum itself, but we go through some of the main terms. *[8]*

3.1.1. Daily meetings

In the scrum it is normal but not mandatory to have daily meetings. These are kept to improve team communication. The main subjects are to find out what people have done yesterday, what they are going to do today and if any new problems have arisen.

3.1.2. Burn down chart

The burn down chart is a chart where on the x axel is the time in days, and on the y axel is the hours of work still to be done. The burn down chart helps to visualize if the project is going to be finished in schedule. This is maybe the most important tool offered by the scrum.

3.1.3. Backlog

A backlog is simply a list of tasks. There are multiple different backlogs depending of the purpose. The backlogs help people to prioritize tasks and to focus on essential.

3.1.4. Product backlog

In every phase of products life cycle there is most probably a list of tasks to be done. Tasks can be any tasks from maintains tasks to implementation tasks. In the scrum all the tasks to be done for one specific product are collected in the product backlog. Some of these tasks may never be done, but those are still in the backlog because we never know if we actually have time later on.

3.1.5. Release backlog

Production is often split to releases. A release is an initial or upgrade version of a software product. The length of a release is normally somewhere between three and twelve months, but there really is not any minimum or maximum length. A release backlog contains all the tasks defined to be done in the specific release. These tasks are chosen from the product backlog. The release backlog is not mentioned in all the resources of scrum, but it is highly useful, and used in this thesis project.

3.1.6. Sprint backlog

This chapter is only a short introduction, and does not try to explain all what is defined around sprints. It is highly recommended to read scrum manuals for more information.

A sprint is the same as an iteration. In scrum, it is a short period of time, normally three to thirty days. A sprint backlog is a list of tasks to be done in this specific sprint. A release contains multiple sprints, and the tasks to sprint backlog are brought from the release backlog. If a next release is planned to be done in six months, and a sprint is defined to be one month long, there will be six sprints in the release.

A sprint is more than just a period of time, it is a well defined production step. When a sprint plan is done, accurate definition of product evolution is done. It measures the progress, did the team get all the sprint tasks done or are we behind the schedule. If the sprint was difficult and the team was able to finish only half of the tasks, we can estimate immediately that we are not able to finish all the tasks in this release.

3.1.7. Product owner

Product owner is a person who is maintaining the product backlog. He is the person who prioritizes the features needed most by the stakeholders. He is responsible that the team is doing what is needed most at the moment.

3.1.8. Scrum master

The scrum master is more or less like a project manager. His main task is to open any blocker situations, like a “broken monitor” situation. There is not just one type of scrum masters, but instead the role is adaptive. If the scrum master is technically strong, he can help with unexpected technical problems. If he is not technically strong he has to be able to find someone to solve the problem.

3.1.9. Subject matter expert

There are rarely position called subject matter expert, this person is simply someone who knows some area really well. It can be some specific software knowledge or knowledge from the area the software is done. These persons are anyway the ones who know the best how long it takes to finish some specific task. In this way they are very valuable for any project and project manager.

3.2. Development process

A development process helps people in the project to work in deterministic way. There are many different process models to choose and there is lot of theory behind the facts why things are done like those are done. There is no reason to force to use some model exactly like it is if it does not fit, but we can still take guidelines. Accurate explanation of different processes does not fit in the scope of this thesis, so only mainlines are shortly introduced. The process used in this thesis is slightly simplified scrum. The scrum is good as it is, but in this project I just did not feel a need for everything.

The product owner and the subject matter expert in the project is the supervisor. The role of scrum master and scrum team is left for me. The daily meetings of scrum are removed. In this project we have only one person working day by day, so there really is not any reason to have one. The project progress and estimate is still checked daily. The burn down charts were not used either, even that those are normally highly useful. The progress of this small project was easy to follow, so drawing charts could not bring any extra benefits.

The product backlog is not used in the thesis project. The focus is all the time kept in the tasks actually done in the thesis. The release and sprint backlogs have more details. A sprint is used to collect all the tasks which belong to specific phase in the project life. In some cases a sprint can contain only one task. It is used to concentrate the focus so that the project goes forward in deterministic way.

4. Software design

This chapter introduces common software design topics, which are important to take care or just good to have in almost any project. This theory is also used as a base for this thesis project.

4.1.1. Maintainability

The maintainability is often forgotten in the design, but in fact this is an important thing to take care. It is rare that software once written will never be touched again. It is also common that there are more people working with the same piece of code, if not with the initial version, but probably in some phase of codes life time. Maintainable software is something where is easy to come back, and it is fast to understand for everyone. An inexperienced programmer does not usually understand or believe that he will forget the code soon after he stops working with it. It is also common that an inexperienced programmer tends to make “smart” hacks or complicated code, almost like proving that he knows how to code. This may even lead to a syndrome where a programmer writes as tricky and complicated code as he can. It is also common, unfortunately, that the management does not understand maintainability, or they understand that the maintainability is important, but they do not know what is affecting to it.

We divide the maintainability issue to following three sections; documentation, clear design, and low inheritance tree.

4.1.1.1. Documentation

Documentation is probably one of the least valued things, but still being one of the most important. Humans tend to see the current situation, but not to think much of the future. It is like smoking, person thinks that is ok until he or she gets a lung cancer. Missing documentation does not kill, but a big project without documentation is heavy for any development team. A good documentation provides fast route to understand the architecture and implementation. It shares the information efficiently through the project group and it decreases the need of specialists. A specialist is a person who knows and remembers the code which others do not, and so comes important key member in the group. Documentation helps new people to get in the project without using time of others, and so decreases the need of specialists. It helps to make estimates of development times, and it makes development faster. If you have a bug in your software which could also be a feature, a good document will clarify it. Sadly in many companies they do not write documentation, and if they have not got used to have one, they do not know how to value it. The budget and time saved by documentation will not be seen immediately, but at the next time when somebody has to dig in the code, it feels like a life saver.

There are many kinds of documents and this thesis takes advantage from two of the most important ones. The first one is the Application Programming Interface (API) documentation and the second one is the design document.

4.1.1.1.1. Application Programming Interface (API)

The most important software project documentation is the Application Programming Interface (API) documentation. This is the one made especially for the developers. The purpose is to help a programmer to

understand how a piece of software is meant to be used, without need to look in the source file. It is useful to write one even if some parts are easy to understand, API documentation also verifies what is wanted. It does not take much time to write a simple API document, and even a simple one is multiple times better than no documentation at all.

4.1.1.1.2. Design document

The second most important is the architecture design document. The document contains information how pieces of software are connected to each other. Once again a short description is multiple times better than no documentation at all. For example an application which contains ten classes and no documentation, it is most definitely time taking to figure out how those are working together. A short description of what those do and why, and how those are communicating with each other, makes the developers job much faster.

4.1.1.2. Clear design

A clear design and implementation, it immediately shares opinions and flames war. Let's drop most of it away and let's concentrate to the humans reading code. Usually humans cannot remember million unorganized lines of code. To handle this we usually categorize the code and separate it to multiple entities, each entity fitting nicely in one source file. Then we separate the functionality and write each one of those in its own function. The function size should be equal or less the size a person can parse and remember easily. This is probably the most important rule from all of those.

More important over the source file issues is the header file, which sums the functionalities which are offered by the source file. A header file is often the only thing read by other developers. This is the reason why a header is more important. There is only one rule for a clear header file, keep it simple. Remove everything that is not mandatory, or transfer into the source file. Remember that the humans are slow to parse code, and more characters and lines there is to parse, more awkward it is to use.

Do not implement anything too far to the future. It often happens that those “maybe needed” features are actually never needed. If you implemented something you needed or you thought you would need, there is no reason to remove it, if it could be still used in some other situation. This is not a statement to say that feature full classes are bad, but instead that a class full of unused features is.

4.1.1.3. Low inheritance tree

Inheritance is one way to reuse code, and reusability is good, so why not then deep inheritance trees? Every class in the tree is more or less static, connected from the top and under. You cannot take anything from the middle, and a tiny change will affect to every class inherited. This makes the whole tree actually static, and leads to monolithic design.

Everyone who has been building class hierarchies, inheritance trees, has ended up to a situation where it is not so easy to decide in which class a feature X belongs. It is also common that in the half way of implementation you have to move a feature in the tree, and change every object related to the feature. It usually needs lot of thinking to get all correctly in a big class hierarchy. Remember, when things get complicated something is probably going wrong.

4.1.1.4. Memory management

The memory management is a huge subject to speak, and there are many books from different memory managers and models. This thesis does not go deep in any manager or model, or neither tries to implement one. The memory management is still important area and should not be forgotten.

The code should be written in way that the memory is freed in logical places and systematically. More memory management you can write off from the hands of class user, the better. In this thesis we use C++ language, which provides a nice feature called auto pointer, which helps exactly in previous.

Another thing, which is actually often forgotten or unknown, is the memory fragmentation. Sure, it is not the most important thing for a programmer to remember, and many programmers do not even care of it. Memory fragmentation hits to the performance of processor cache and to the used memory manager. A memory manager in a project is usually centrally handled, and the main improvement done is that, it does not let the memory to fragment.

The hit to the cache performance comes from the fact that it is hard for a processor to predict which piece of memory will be needed for next. This increases the amount of cache misses, which directly affects to the performance.

4.1.2. The PIMPL idiom

The PIMPL stands for “Private Implementation” or “Pointer to Implementation”. The main purpose is to decrease project compiling time, but it has multiple other benefits as well.

The main idea is to move all private members to a local class, to a class which is in the source file. All internal declarations are done in the source file. If you need more private members you add those to the private class in the source file. You only modify the header file when you need to change the API.

Some say that the code comes more complicated, but when compared to the things it simplifies, it is well worth it. For next the pros and cons we achieve with this. The last two sections in this chapter show different implementation options.

4.1.2.1. Compile time

Modifications to private implementation do not initiate recompilation of every file. Only API modifications initiate recompiling. If you need a new private member, you add that to the private class. Only this one source file has to be recompiled.

4.1.2.2. Data protection

The data is protected, encapsulated well.

4.1.2.3. Clear header

As most of the definition goes in the source file, the header is bare and clear. The header file now defines the API, but not internal implementation. The header is the one read by other developers, and the header without private implementation is just easier and faster to use.

4.1.2.4. Application Binary Interface

The ABI is more stable, because every implementation change does not affect to it. Example: If a developer optimizes the implementation and needs a cache pointer, this new member variable is declared in the private implementation class. This does not change the binary size of the public class, and the ABI stays untouchable.

4.1.2.5. Performance hit

It is true that the performance is not as good in all cases. This is mainly because it uses more indirect references. The performance hit is minimal and does not matter really, not even in highly optimized code.

4.1.2.6. Implementation: The PIMPL pointer

From our two implementation examples this is probably the most common way to implement, and that is why we introduce it first. The implementation is simple. In the source file we declare a class, which will contain the private implementation. In the header file we add a forward declaration to the private class, and in the public class we add a pointer to the private class.

```
/* file - source.c */

class Apimpl
{
        /* Our private implementation */
};
```

```
/* file - header.h */

class A
{
private:
        class Apimpl;
        Apimpl *pimpl;
};
```

The only negative sides from this implementation are, the extra space used by the pointer and the memory fragmentation.

4.1.2.7. Implementation: Inheritance from public virtual class

This implementation is based on a pure virtual class declared in the header file. This class defines the interface, the API how the class shall be used. Also it is mandatory to declare a newObject function in the header file. The rest of the implementation goes into the source file, where we declare the private implementation class inherited from the virtual base class. The newObject function creates the object from the private class and returns it as a pointer to virtual base class.

```
/* file – source.c */

class Apimpl : public A
{
        /* Our private implementation */
};


/* file – header.c */

class A
{
        /* Our public implementation */
};

A newA (void);
```

The negative side of this implementation is that the inheritance from this class is impossible. Even that this sounds like a radical flaw, it is not really that radical. If you calculate the times you had to inherit from some class and your architecture was not built in high inheritance tree, you probably notice that it is not that big flaw. If you really have to inherit from some class, this implementation is not to be used.

From the positive side, this does not add any pointers, nor creates any extra memory fragmentation. The implementation is a bit clearer one too. This is the implementation style used in the code written in this thesis.

4.1.3. C++ templates

Templates are powerful but when over used, the consequences can be fatal. Create templates only in cases where the implementation does not require anything from the used type. As a rule of thumb, create a template from container and auto pointer classes only.

4.1.4. Compiling time

Compiling time may not seem to be so important, but especially in large projects it is. If the issue is properly taken care, compiling time can be many times faster. This thesis introduces three ways to improve compiling time.

4.1.4.1. The PIMPL idiom

As introduces before, the PIMPL structure does not change the public interface so often. This can dramatically decreases the need to recompile in rapid development.

4.1.4.2. Include guard

This means that every header is #included only once. This is done by using so called include guards. There are internal and external include guards, which work slightly differently. If the compiler implements internally the

external include guard, the internal include guard is faster. If this is not internally implemented, manual use of external guard is faster. GCC and Cygwin environment are implementing the external guard internally, so only use of internal guard is needed.

```
/* file – internal.h */

#ifndef __INTERNAL_H__
#define __INTERNAL_H__

/* Definitions in the intern.h */

#endif /* __INTERN_H__ */
```

The internal include guard skips the definitions if the file has been introduced already.

```
/* file – external.c */

#ifndef __EXTERNAL_H__
#include “external.h”
#endif
```

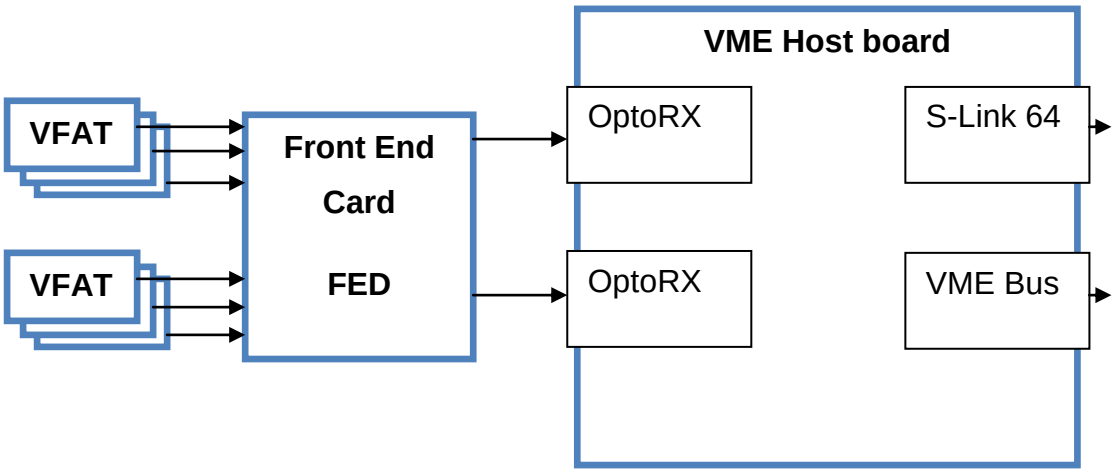
The external include guard goes one step further and it checks the need for header already in the source file. The additional performance gain comes, because the compiler does not have to find and open the file for reading. If the external include guard is implemented in the compiler, the compiler automatically checks if the file has been already read. This thesis does not go deeper in the subject, but it is easy to find more information from internet in case needed.

4.1.4.3. Forward declaration

Include a header when you have to use the API in it, but if you only refer to the class in the header, use forward declaration.

5. The emulator

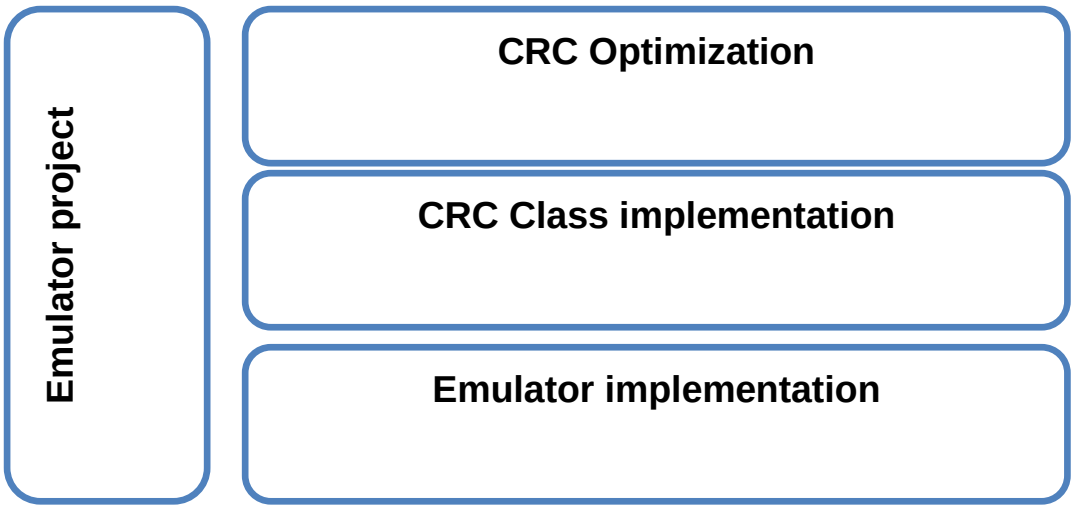
The goal of the emulator project is to make the development of data acquisition software possible without a real hardware. The hardware to be emulated consist VFAT chips, Front End Card (FED) and VME host board. The VFAT chips are the ones reading the information from the sensors. The FED then reads the information from the VFAT chips. In the FED the electronic signal from VFAT chips is changed to optical signal, which then goes to VME host board. On the VME host board, the optical signal is read by an OptoRX card. From the VME host board the packet continues through S-link or VME bus. The packet leaving either from S-link or VME bus, is the packet the emulator produces. The hardware layout is shown in the picture 5.1.



Picture 5.1: The hardware to be emulated.

The project is divided in three separated subprojects. The thesis contains also schedule chapter. The schedule chapter is project management chapter, concentrating to releases and sprints. The schedule is reviewed first and later comes the real implementation chapters. In the first subproject the CRC algorithm is optimized, and the results are verified by benchmarking the old and new algorithms. In the second subproject a CRC class is created,

which uses the information gathered in the previous sprint. In the third subproject the emulator class is created.



Picture 5.2: The subproject. The timeline goes from top to bottom.

The last thing before moving forward is to introduce the idea of simplicity. Everything in the thesis has been done simplicity in mind, even the process and management. Remember still, simple is not the same as easy.

"Keep It Simple, Stupid" – Some unknown developer

6. Project schedule

The chapter contains how the project schedule was built and success level estimations. The schedule is done only from the implementation subprojects. There are no technical details or requirements in this chapter. Technical information is left for the later chapters. A tool called Trac was used with the Agilo plugin to track the progress of the project. Trac was chosen because it is commonly used. The agilo was chosen because it provides additional tools for agile development.

The budget is enough for six months, which is the same as the size of master´s thesis and that creates also the deadline. There is no project post management in the thesis, like bug tracking. The post management is left for the next CERN developers.

The first month was used to get information of environment, and parties working in it. This period was important because it helped to understand where and why this work is needed. This period also opened the terminology to understand what people in the experiment were talking. The period was built from courses, different meetings, introduction tasks and from opportunities to see the real hardware. Also hardware installations were done, like installation of a quarter of T2 detector. This month is shortly described in the introduction chapter.

In the second month, planning of the thesis began with more details. The scope and the objectives were defined. The project definitions from the second month are part of this chapter. Also the details of the project management for the whole project are in this chapter. An initial schedule was created, with a big picture view to the project. The last four months were more interesting from the thesis point of view and are separated under own sections.

In the tables which show the sprints done, the duration of a sprint does not mean the days of work spent to the sprint. Instead it means how many working days there are between the start and end dates. The possible holidays

are calculated in the duration as well. The real time used is clarified in the sprint stories. Only the implementation sprints contain the separated task lists.

As will be seen, there are study sprints which are more or less one task sprints. The study sprints would not be normally as their own sprints, but because in this case it is impossible to estimate the implementation sprint, before the study sprint is completed. Normally the study sprints are contained in some earlier sprints.

6.1.1. Emulator project release

This is a release where goes all the management tasks and tasks which do not belong to any specific release. The release begins 23.03.2009 and ends 31.07.2009. The days before the day 23 were more or less static and I was not personally managing this period. The period contained introductions, meetings, courses related to the CERN, TOTEM experiment and this thesis (like. security course). The tasks and sprints done in this release have least documental value, and that is the reason why this release is mostly scoped out from this document. The following table shows the sprints done in the emulator project release.

Sprints	Start	End	Duration
Setup environment	13.04.2009	17.04.2009	5
Project management	02.07.2009	08.07.2009	5

Table 2.2.1: The sprints of the emulator project release

6.1.1.1. Setup environment

The goal of the sprint was to setup the software environment. The sprint combined studies of what should be installed, and the installations. The used computer was so new that it was not possible to install the scientific Linux, without spending lot of hours. Ubuntu was installed with the XDAQ. The XDAQ is only designed to work with the scientific Linux, so problems were encountered but all began to work in the end. Later a virtual machine was used, as it should have been used in the first place. This move would have saved lot of time.

6.1.1.2. Project management

The status of the project was carefully checked, and it seemed that the time was running short. Big part of project management related information was sorted, filtered and documented. This chapter is part of this documentation.

6.1.2. Optimize CRC algorithm release

The goal of the release is to optimize CRC algorithm. The following table shows the sprints completed in this release.

Sprint	Start	End	Duration
CRC Studies	23.03.2009	03.04.2009	10
CRC Studies 2	06.04.2009	10.04.2009	5
CRC prototype and benchmark	20.04.2009	08.05.2009	15
CRC Optimization documentation	11.05.2009	22.05.2009	10

Table 5.1.2: The sprints of the CRC Optimization release

6.1.2.1. CRC studies

The whole domain was new so it was impossible to estimate accurately how long it would take to study it. Two weeks were reserved and in the end of the sprint things were still unknown. The amount of information around CRC algorithm was underestimated.

6.1.2.2. CRC study 2

CRC studies sprint was too short, so another study sprint was initiated. This time one week was reserved, but one day from this was not needed to be used to the sprint. The sprint succeeded, and the optimization studies were completed.

6.1.2.3. CRC prototype and benchmark

First a prototype was written in C, and tested that it can produce correct CRC calculations. After this it was benchmarked and a small library was built. Initial plan of the CRC class was built as well. For this period there was lot of issues from outside of the sprint and because of that the estimate was set to 15 days. There was ten days of active sprint hours. The optimization was success and clear performance improvement was achieved.

6.1.2.4. CRC optimization documentation

This was straight forward documentation sprint. A part of this documentation is included in the thesis. This sprint contained 8.5 days of actual work.

6.1.3. Implementation of CRC Class

The goal of the release is to create a CRC class. The following table shows the sprints completed in this release.

Sprint	Start	End	Duration
C++ Studies	22.05.2009	26.05.2009	3
CRC Class	27.05.2009	04.06.2009	7
CRC Class documentation	08.06.2009	12.06.2009	5

Table 2.2.3: The sprints of implementation of CRC Class release

6.1.3.1. C++ studies

This did not include much syntax studies, but instead how to use C++ to create a good design and implementation. This was very educational and very successful sprint. The chapter software design has lot of old experience, but it also shows some of the fruits collected in this chapter.

6.1.3.2. CRC class

This was the implementation sprint. The design task in the sprint is a task from where design hours were taken slowly while the project was getting forward. The following tables are showing the initial estimate and the later one the real hours spent.

CRC class design	16.0h
RefCount class	4.0h
ref_ptr class	10.0h
Table class	10.0h
CrcChecker class	24.0h
Benchmark	8.0h
Totals:	**72.0h**

Image 2.2.3.2.1: CRC Class task list with estimates

CRC class design	16.0h
RefCount class	3.0h
ref_ptr class	6.0h
Table class	12.0h
CrcChecker class	24.0h
Benchmark	6.0h
Totals:	**67.0h**

Image 2.2.3.2.2: CRC Class task list with real hours spent

In the later chapters there is AutoMap class which is not in this list. This is because the need for this class was found in implementation phase. AutoMap hours are included in the CrcChecker class. There was a design flaw which was fixed, but luckily the sprint was success also with the additional work.

6.1.3.3. CRC class documentation

This was straight forward documentation sprint. The estimate for this sprint was five days but it was finished in four days.

6.1.4. Implementation of emulator class

The goal of the release is to implement the emulator class. The release is the last one before the deadline, and the schedule was getting really tight. The following table shows the sprints completed in this release.

Sprint	Start	End	Duration
Study the data acquisition packet frames	09.07.2009	15.07.2009	5
Implementation of the emulator class	14.07.2009	24.07.2009	9

Table 6.1.4: The sprints of Implementation and integration of emulator class release

6.1.4.1. Study the data acquisition packet frames

The goal of this sprint was to learn how the packets of data acquisition are constructed, and which part of the data has to be emulated and how. The information was collected through meetings and using documentation provided by CERN. [16][17] The sprint was two days shorter than it was estimated.

6.1.4.2. *Implementation of the emulator class*

The goal of the sprint is to implement the frame classes and emulator classes. There were four separated tasks in the sprint. See the task estimates from the following table, and the real hours spent from the second table.

Task	**Estimate**
Implementation of hor2ver	8h
Implementation of vfatgenerator	24h
Implementation of optorxemulator	20h
Implementation of totfedemulator	20h
Total:	**72h**

Table 6.1.4.2.1: Estimated hours.

Task	**Real hours**
Implementation of hor2ver	8h
Implementation of vfatgenerator	22h
Implementation of optorxemulator	18h
Implementation of totfedemulator	16h
Total	**64h**

Table 6.1.4.2.1: Real hours spent.

The estimation was enough this time as well. It is actually rare that the estimates are well succeeded, it is really hard to make reliable estimate. Usually the projects are late, or there are long hours before the deadline. One part why the estimates were so good is probably because this was done as a study. There were more time and also discussion how things should be done, compared to normal industrial way. A one good point here is anyway that all the tasks are small, none is measured as a one week long tasks. It is just easier for us to measure small tasks than large ones.

7. Optimization of CRC Algorithm

The Cyclic Redundancy Check (CRC) is an error detection technique invented by W. Wesley Peterson in 1961. Error detection is important when transferring data through noisy channel and we want to be sure that the data was not corrupted. To achieve this we calculate a value representing the data, and we append the value to the end of the sent data. The receiver calculates the value from the data it received in the same way, and compares that to the value calculated by the transmitter. If the values match, the data is assumed to be correctly received.

There are many different techniques to detect errors and none are bullet proof. The main problem is that from multiple different data streams we can get exactly the same checksum. Usually an error detection technique is evaluated by how fast it is to execute and how error proof it is. The CRC is currently popular because it is easy to implement in binary hardware, mathematics are simple, and it has been proven to be efficient in error detection. The IEEE recommended 32 bit CRC in 1975 and now it is widely used, like in Ethernet networks.

7.1. Primitive error detection

This chapter introduces a primitive error detection technique. The reason is to show a simple example why none of the error detection techniques can be 100% reliable.

The simple technique sums all the bytes in a message in a byte wide register and the total is the checksum. As we can understand, multiple different combinations can produce a same checksum. The byte position, total length of data and value could change in many ways, and we still could have the same checksum. As an example we have three bytes:

Data:	2 14 8
Data with checksum:	2 46 8 56
Erroneous data:	2 47 8 56
Undetected error:	2 8 46 56

The data is the data to be transmitted. The data with checksum is the data with checksum appended to the end of it. The underscored byte is the checksum byte. In the erroneous checksum case we can see how the checksum would detect an error because 2+47+8=57, instead of 56. The undetected error is the received data which is corrupted, but the checksum is still valid and the error would go undetected. As we can see, an error of n bytes long can go undetected if there is just one byte with a compensating error.

By using a wider register, like 16bits, we could get better result because the register would not overflow so fast. The sum algorithm would still fail, but one byte long value could not compensate an error any more. Instead it would need 16bit value to compensate.

If the message is short it does not matter how wide the register is, because the summed value would never get big enough to touch the high end bits. It would leave all the high end bits to zero. One byte can touch eight bits and more only when the summing is overflowing. To get better error detection efficiency, one byte should touch the register from wider area.

As we can see, the simple algorithm would not do it in the modern world. What we can conclude from this chapter is that, to get more efficient error detection we need complexity. If we begin to think how we could achieve this, we notice that there are many different options coming in mind. Two ways could be, to use a wider register, and to get more randomness from each byte. We do not go any deeper to different kind of possibilities and instead we move directly to the CRC theory.

7.2. CRC theory

To understand the CRC it is good to understand how the idea evolved. In this chapter we go through the binary division, which is probably the base for all modern error detection algorithms. Secondly we preview polynomial arithmetic, which describes the evolution from division to exclusive or (XOR) operation, which is used by CRC algorithms. In the third part we go through a property of XOR operation. The last part is introduction to term poly, which is rather important term to know when working with CRC algorithms.

7.2.1. Binary division

The idea behind CRC algorithms stands on division operation where the remainder is the checksum. This is clearly more efficient in error

detection than summing. To get good results with division, the divisor has to be more or less as wide as the register is. Another important point is that we do not divide every byte in a message, but instead we divide the whole message as a one big number. The divisor is a well defined, fixed number.

The following listing is an example of binary division, using the same bytes as in our summing example. A byte wide register is used to keep the example as simple as possible. The divisor in our example is binary 1001, which equals nine as a decimal number. The procedure itself is the same as in decimal long division.

2 = 0000 0010

46 = 0010 1110

8 = 0000 1000

dividend = 0000 0010 0010 1110 0000 1000 == 10 0010 1110 0000 1000

divisor = 1001

```
              11111000000000        QUOTIENT
            ________________________
1001    ) 1000101110000001000        DIVIDEND
          0000
          ------
          10001
          -1001
          =====
           10000
           -1001
           =====
             1111
            -1001
             ====
              1101
             -1001
              ====
               1001
              -1001
               ====
                0000
                0000
                -------
                 0000
                 0000
                 -------
                  0000
                  0000
                  -------
                   0000
                   0000
                   -------
                    0000
                    0000
                    -------
                     0001
                     0000
                     -------
                      0010
                      0000
                      -------
                       0100
                       0000
                       -------
                        1000
                        0000
                        -------
                        1000        REMINDER
```

7.2.2. Introduction to polynomial arithmetic

When working with the CRC or reading some manual of it, you cannot escape from word polynomial. The manual itself may point that the polynomial is an important word. Normally in the manuals it says that the specific CRC algorithm is using particular polynomial. Also the manuals point out that the CRC algorithms are using polynomial arithmetic. We do not need to know all from polynomial arithmetic, and instead we go this chapter through with an example from the parts we need.

As previously was seen, the message byte 46 in decimal was 0010 1110 in binary. This same value in hexadecimal would be 0x2E and in polynomial representation it would be:

0*x^7 + 0*x^6 + 1*x^5 + 0*x^4 + 1*x^3 + 1*x^2 + 1*x^1 + 0*x^0

Each bit is represented as a bit value multiplied by x squared the bit position. If we remove the zero terms and multiplications by one, we get:

x^5 + x^3 + x^2 + x^1

So let's calculate something with the polynomial system. To keep the example simple we take two small values, 11 and 3, and we multiply.

11= 1011

3 = 0011

```
(x^3 + x^1 + x^0) * (x^1 + x^0)
= (x^4 + x^3) + (x^2 + x^1) + (x^1 + x^0)

= x^4 + x^3 + x^2 + 2*x^1 + x^0
```

Because we are calculating binary values, we know that the x is two. With this knowledge we can then calculate the binary carries as well. This means that the 2*x^1 equals x^2, and after we have calculated those all, we get:

= x^5 + x^0

Let's make then a scenario where we do not know that the x equals two. In this scenario we cannot assume that 2*x^1 equals x^2, and we cannot then handle the binary carries either. Now every value is isolated from each other. The mathematicians tried to solve this by changing the polynomial rules. One of these attempts is called polynomial arithmetic mod 2, which is actually also used in the CRC. In these rules all the coefficients are calculated mod 2, which mean coefficients are either 0 or 1. The second rule is that there are no carry. These rules are actually the same as in binary arithmetic without carry. The rules in mind the previous calculation would simply turn out to:

x^4 + x^3 + x^2 + x^1 + x^0

Another important point with the new rules is that, the addition and subtraction operations would turn to the same operation. For example:

```
 1011   1011
+0110  -0110
=====  =====
 1101   1101
```

The subtraction/addition truth table:

```
0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0
```

The truth table is actually exactly the same as the exclusive or (XOR) operations truth table, and this is where we come to the CRC. We use modulo 2 division to divide the messages in the CRC arithmetic. The last important things to know between binary division and mod2 division are that, if the most significant bit is one the divisor divides in. Secondly, equally many zeroes will be appended to the end of the dividend as the divisor is wide. The zeroes in the end make sure that all the bits in the message will affect to the result. In the following listing we use the same values as in the binary division and the new mod 2 division rules:

```
            1111100000000000000       QUOTIENT
          _________________________
1001    ) 1000101110000001000 0000    DIVIDEND
          1001
          -------
           0011
           0000
           -------
            0110
            0000
            -------
             1101
             1001
             ====
              1001
              1001
              ====
               0001
               0000
               -------
                0010
                0000
                -------
                 0100
                 0000
                 -------
                  1000
                  1001
                  ====
                   0010
                   0000
                   ====
                    0100
                    0000
                    ====
                     1001
                     1001
                     ====
                      0000
                      0000
                      ====
                      0000            REMINDER
```

We did end the calculation before end, because the rest of the bits are zero and cannot affect to the result. This is just to fit the calculation on one page.

7.2.3. Interesting property of XOR operation

In the mod 2 division we have multiple XOR operations one after another. What is the most important thing for us to notice is that the order of the XOR operations does not matter. If we XOR four different values, it does not matter which one is XORed first or last. For example, let's XOR five numbers; 1, 2, 3, 4 and 5:

```
    0001
xor 0010
xor 0011
xor 0100
xor 0101
========
    0001
```

Reordering the numbers to 2, 4, 3, 1 and 5:

```
      0010
xor   0100
xor   0011
xor   0001
xor   0101
========
    0001
```

7.2.4. Poly

The name comes from polynomial arithmetic, and often a CRC algorithm is said to use a particular polynomial. The divisor is also called as generator polynomial, which then turned out to simple polynomial. In the CRC talk this word polynomial then turned to even simplified poly. The poly is the divisor in CRC arithmetic. It is really important part of CRC algorithms, and almost solely responsible how efficient a particular CRC algorithm is. There are

books written from this subject, where professional mathematicians proof why a particular poly is an efficient one. The mathematics of poly is out of scope from this project. A poly is usually expressed as a hexadecimal value. The VFAT chip at CERN uses 16 bit poly and its value is 0x8408.

If you now calculate long binary mod 2 division, with a poly found from some source file and compare that to the CRC algorithm result, you notice that the result is not the same. The poly from source code actually contains number one at front of it. Like 0x8408 would turn to 0x18408. This number one is actually omitted by the algorithm implementation. To show why and how this happens, we have to implement something. In the next chapter we implement a basic algorithm and explain how this number is omitted.

7.3. Basic CRC algorithm

In the first part of this chapter we see an implementation of basic CRC algorithm in C language. The algorithm is 32 bit, because that is the most standard. On the same time, we go through why a poly loses its first bit one in CRC algorithms. In the second part we make the first optimization to the basic algorithm.

7.3.1. Basic algorithm

The basic algorithm follows pretty accurately the long binary division mod 2.

```
for (i=0; i<len; i++)
{
          byte = *(data++);

          for (j=0; j<8; j++) {
              if ((byte >> 7) ^ (reminder >> 31)){
                   reminder = reminder << 1 ^ POLY;

              } else {
                   reminder = (reminder << 1);
              }

              byte <<= 1;
          }
}
```

Listing 5.1: The basic CRC algorithm

The basic algorithm reads every byte from the data in a loop and in a second loop goes through every bit in the byte. In the inner loop the algorithm checks the result of XOR operation between the first bit of the data byte and the first bit of the reminder. If the statement is true, we shift the remainder one bit and XOR the poly in it. If the statement is false, we shift the bit out from the reminder. As a final thing we shift the next bit from byte to be the new top bit. One of the important things is that the top bit of the POLY, XORed with the top bits of data and reminder, decides if we XOR at all. Then why the top most bits are XORed and shifted out, before being XORed with the POLY at all. Here comes actually the extra bit one to the beginning of the POLY. The same extra bit which is not shown in the CRC algorithms, but which is visible when calculating on paper. The algorithm knows that this bit is there, so it is enough to XOR only the data and the reminder inside the if-statement. We do not either lose information bits (bit ones), because when the extra POLY bit is XORed with shifted out bit one, it XORs to zero.

In the following example we use four bit register to calculate the reminder, and it has some previous calculation data in it already. The rest of the data is zero, so when shifted there is only zeroes left. The out shifted bit is separated with space, but still visible in the example. The poly is five bits when the extra bit one is calculated in. The extra bit is underscored:

```
DATA:       0110

POLY:    1  1001
REG:       1101


Shift REG and DATA:

DATA:    0  1100

POLY:    1  1001
REG:     1  1010


XOR the high bits (the space separated bits):

DATA:    0  1100
REG:     1  1010
XOR:     1  1010


XOR the POLY and the REG:

POLY:    1  1001
REG:     1  1010
XOR:     0  0011
```

7.3.2. Optimized basic algorithm

The basic algorithm puts bits in the register from the right end and at the same time shift the highest bit out from the left end. If the register is 32 bit wide, it means that before the first 32 bits, no bit it shifted out. This also means that nothing is XORed while shifting the first 32 bits. After someone had noticed this, he loaded the register with the first 32 bit of data before starting the loop.

Let's go through one simple modification more before going to the next example. In the if–statement of basic algorithm we XOR the next out shifted bits and check if the result is one. To get the same result, we can XOR

the data bit to the reminder register and then compare the highest bit of the register. This does not optimize much, but it definitely gives a cleaner implementation.

```
register = *data++ << 24;
register |= *data++ << 16;
register |= *data++ << 8;
register |= *data++;
len -=4;

for (i=0; i<len; i++)
{
    byte = *(data++);
    for (j=0; j<8; j++) {
        if (register & 0x80000000) {
            register = (register <<1) ^ POLY ^ (byte >> 7);

        } else {
            register = (register << 1) ^ (byte >> 7);
        }

        byte <<= 1;
    }
}
```

Listing 5.2: The optimized basic CRC algorithm

7.4. Table-driven algorithm

The chapter is divided in two parts. The first part introduces the basic table-driven algorithm, and the second part introduces slightly improved table-driven algorithm

7.4.1. The basic table-driven algorithm

In the CRC algorithms the bits which are one in the data, define when to XOR the poly. If we look the long binary division mod two, we can see that the reminder in the end is only the poly XORed multiple times with different offsets to the data. We can know the next XOR operation position after another, because every XOR operation is modifying the reminder. But what if we already knew the offsets, we could then XOR the poly even in random order and we would get the same result. In the next example, we XOR short binary value 101 every time when the high bits are one, until the end of message.

```
DATA: 1101 1011 000
      101
      ===
      0111 1011
       101
       ===
       010 1011
        10 1
        =======
        00 0011 000
             10 1
             ====
             01 100
              1 01
              ====
              0 110          RESULT
```

Let's put the data aside for now and XOR together the polys with the same offset as in the previous example.

```
          101
           101
            10  1
                  10  1
                   1  01
          ============
          1101  1011  110
```

If we now XOR this value with the data, the result would be the same 110. Actually what we know now also is that, when the data is 11011011 and the poly 101, the result will be always 110. If we now calculate a result for every byte and we put the values in a table. Then when we don't ever have to XOR bits in the byte, and instead we can just check the result from the table. This is where we get to the implementation of table-driven algorithm.

```
/* Table creation algorithm */

for (i = 0; i < 256; i++)
{
    crc = i << 24;
    for (j = 0; j < 8; j++)
    {
        if (crc & 0x80000000)
            crc = (crc << 1) ^ POLY;
        else
            crc = crc << 1;
    }
    table[i] = crc;
}


/* Table-driven algorithm */

result = *data++ << 24;
result |= *data++ << 16;
result |= *data++ << 8;
result |= *data++;
len -= 4;

for (i=0; i<len; i++)
{
    result = (result << 8 | *data++) ^ table[result >> 24];
}
```

Listing 7.1: The basic table-driven algorithm

This algorithm is actually rather fast, and you can find similar implementation from many places. A table for each byte has 256 items, and 32 bit algorithm needs four bytes per each item. 4 bytes x 256 makes 1KB of memory, and this is not much for today's computers.

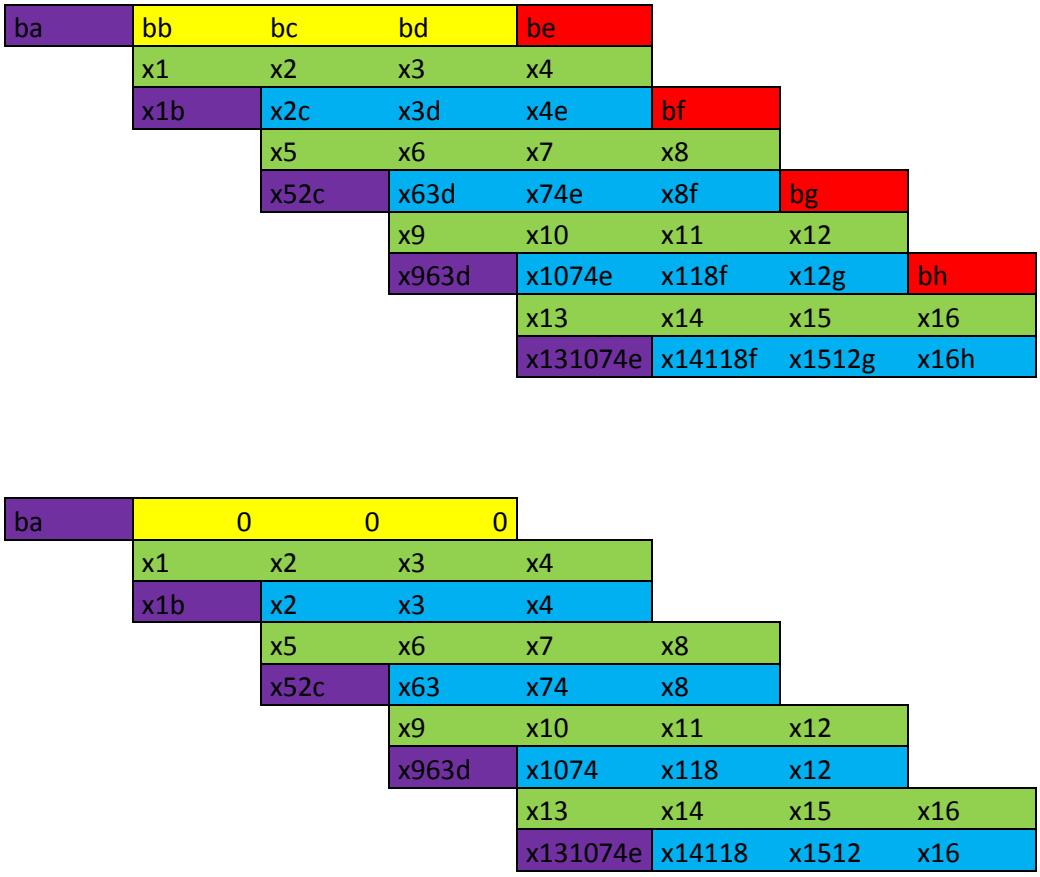
Some people are creating even a table for each 16 bits. Would this make it double as fast, because every time you read a value for 16 bits of data. In this case the table size would be 65536 items and 4 bytes x 65536 makes 256KB of memory. 256KB of random access memory (RAM) is not much, but we would like to fit that in the cache of processor. 256KB would not fit in the L1 cache and it would flush the L2 cache, if not totally, a big part of it.

In this project we do not test processor cache efficiency, and instead we estimate that the eight byte table is the best one. This will most probably change when the cache sizes grow.

7.4.2. Improved table-driven algorithm

In the table-driven algorithm we shift a byte out from the left and a new byte in from the right side of the register. In this way the bytes go through the register. If we now remind us from the property of XOR, that it does not matter in which order we XOR values together. In this case it would not matter do we XOR the new byte immediately when there is space in the register, or in some later phase. We have to just take care we XOR the new byte in the register before or at the same time, as it is time to shift the byte out from the register.

In the following tables ‘b’ means the same as data Byte and letter next to it, is there for an index numbering. An ‘x’ means XORed poly byte, and number next to it, is again just index number to separate the XORed bytes. The cells with a yellow background describe the register with initial data. The green cells are the poly. The blue cells are the register after XOR operations. The red cells are the new data bytes coming from the right side. The violet cells are the cells which will be shifted out for next. Notice that in the second table there are not red cells because the new data byte is directly XORed to the violet byte.



As we can see the violet bytes are the same in the end, when the bytes are shifted out. This means that we do not have to drag the data bytes through the register. Instead we can XOR the byte just before we use it to fetch the next value from the look up table.

```
while (len--) {
    result = (result<<8) ^ table[(result>>24) ^ *data++];
}
```

Listing 7.2: The improved table-driven algorithm

There are two improvements through this. We can remove the lines to initialize the register. Secondly, when the last data byte is in the register, we do not have to begin to feed extra zeroes from the right like before. Feeding the extra zeroes from the right end can be actually a heavy task to complete. In some implementations this means even reallocation of memory

and adding manually those zeroes to the end. This kind of implementation would indeed be heavy. With this improvement we can be sure that there is no need for awkward implementations to add the zeroes.

7.5. 32 bit table-driven algorithm

The name 32 bit table-driven algorithm might be a bit miss leading. 32 bit means here the amount of bits of data read, per each round of algorithm's main loop. The previous algorithms read one byte of data at time. This means that if we can do it 32 bit at time, the algorithm will have three memory-to-register operations less. The only negative components are that the amount of data has to be dividable by four, and we have to take care of processor endianess. To keep the following example simple, there are no endianess checks in it.

```
while (data<end)
{
    result ^= *data++;
    result = table [result >> 24] ^ result << 8;
    result = table [result >> 24] ^ result << 8;
    result = table [result >> 24] ^ result << 8;
    result = table [result >> 24] ^ result << 8;
}
```

Listing 7.3: The 32 bit table-driven algorithm

7.6. Current CRC algorithm

The current algorithm is more or less direct copy from the binary hardware calculation mechanism. Each bit in each byte is looped around and XORed by the rules of CRC. In the algorithm the topmost bit is the rightmost, and so the shifting is done to the right instead of left like in previous theory. The algorithm begins to go through the stream of data also to opposite direction.

```
inline static unsigned short int
crc_calc (unsigned short int crc_in,
                  unsigned short int dato)
{
          int i;
          unsigned short int v = 0x0001;
          unsigned short int mask = 0x0001;
          unsigned short int d=0;
          unsigned short int crc_temp = crc_in;
          unsigned char datalen = 16;

          for (i=0; i<datalen; i++){
              if (dato & v)
                   d = 1;
              else
                   d = 0;
              if ((crc_temp & mask)^d) {
                   crc_temp = crc_temp>>1 ^ 0x8408;

              } else
                   crc_temp = crc_temp>>1;

                       v<<=1;
          }

          return(crc_temp);
}
```

```
unsigned short
cern_algorithm (const void *data,
                            unsigned int len)
{
         int i;
         unsigned short int crc_fin = 0xFFFF;
         const unsigned short int *sdata;
                  sdata = (unsigned short int*)data;

         for (i=len/sizeof(unsigned short int)-1;
                i>=0; i--) {
             crc_fin = crc_calc (crc_fin, sdata[i]);
         }

         return crc_fin;
}
```

Listing 7.4: The current CRC algorithm

The current algorithm actually looks clumsier than the bit-by-bit basic algorithm in the previous chapter. The function `cern_algorithm` goes through the whole message byte-by-byte and calls `crc_calc` for each of the bytes. The `crc_calc` function goes through each byte, bit-by-bit. There are two if-else-statements from which the first one checks the topmost bit in incoming data and second one handles the register compared to top bit separately. This double stepping does visualize the algorithm really well, but it also needs some extra instructions. These extra instructions are not lethal, and most probably in the compiler optimized code the result is more or less the same as in the basic algorithm we saw earlier.

7.7. Optimized CRC algorithm

The old algorithm is using basic CRC algorithm with a bit clumsy implementation. It is clear that there is much to improve. The base of the new algorithm will be the 32 bit table-driven algorithm. This type of 32 bit algorithm had the least C operations, so it is a good base to start.

7.7.1. 16 bit table-driven algorithm

There are two things to modify from 32 bit algorithm to get it work as 16 bit CRC algorithm. First, the reading operation has to be modified to read 16 bits at time. Secondly, the order we read the bytes and bits has to be reversed to support the CRC algorithm.

```
unsigned short
crcr_algorithm (const void *data, unsigned int len)
{
          const unsigned short *end;
          const unsigned short *ptr;
          const unsigned short *lut;
          unsigned short reg;

          end = (const unsigned short*)
                              (((unsigned int)data) + len);
          ptr = (const unsigned short*)data;
          lut = (const unsigned short*)(table);
          reg = 0xFFFF;

          while (ptr < end) {
              reg ^= *--end;
              reg = (reg >> 8) ^ lut[reg & 0xFF];
              reg = (reg >> 8) ^ lut[reg & 0xFF];
          }

          return reg;
}
```

Listing 9.1: First implementation of the new CRC algorithm

7.7.2. Assembler optimization

The new algorithm looks good and it seems to have the smallest C footprint from all the algorithms above. It would still be interesting to see if we can do something for it from assembler level. Sure there is no sense to write assembler these days, even if the implementation would be faster. But what we can do is to see if we can get ideas how to optimize our C code.

In this project we use gcc version 4.3.3. The gcc can make us an assembler output from the C code by using flag "–S". The default assembler output syntax is AT&T, which is a bit messy compared to the Intel syntax. This is of course every ones personal opinion, but it seems that most of us think that the Intel syntax is more comfortable. With extra parameter "-masm=intel" we can request assembler in Intel syntax. We use "–O3" optimization flag also, because we do not want to work on anything what compiler could do for us. The whole command will then be:

```
gcc *.c –S –O3 –masm=intel
```

You can see the whole assembler output from the attachment XXX. For optimization reasons we are mainly interested to see the algorithm's main loop. The following assembler listing will have additional comments to clarify it, which will not be found from the attachment.

```
.L6:
        sub     ecx, 2              ; --end
        xor     ax, WORD PTR [ecx]; reg^= *end;

        mov     edx, eax            ; copy for reg>>8
        movzx   eax, al             ; reg&0xFF
        shr     dx, 8               ; reg>>8
        xor     dx, WORD PTR [ebx+eax*2]
        mov     eax, edx            ; copy for reg>>8
        movzx   edx, dl             ; reg & 0xFF
        shr     ax, 8               ; reg >> 8
        xor     ax, WORD PTR [ebx+edx*2]
        cmp     esi, ecx            ; (ptr < end)
        jb      .L6
```

Listing 9.2: 16 bit table-driven algorithm assembler output

From the listing we can see that the main calculation lines of algorithm are each four instructions long. The difference between the C lines in assembler is that, on each C line we use different registers to do different operations. Like on the first C line we use dx for shifting and on the second C line ax. Here is the first C calculation line separated from the rest:

```
        mov     edx, eax                    ; copy for
reg>>8
        movzx   eax, al                     ; reg & 0xFF
        shr     dx, 8                       ; reg >> 8
        xor     dx, WORD PTR [ebx+eax*2]
```

Listing 9.3: One C calculation line separated

On the first line we copy the reg, so that we can operate on two different registers. We need two registers to hold two different reg calculations, one for shifting and another for ANDing. Other one goes to edx and other one to eax register. On the next two lines in the middle, we calculate the shift and, AND operation. On the last line we fetch from the look up table and XOR it all together.

The only thing that really hits in the eye is the first '`mov`' operation which is simply copying the data. After this we then operate with '`movzx`' which could also make a copy at the same time. If we now remove the '`mov`' instruction and reorder the registers:

```
movzx     edx, al                                ; reg & 0xFF
          shr       ax, 8                        ; reg >> 8
          xor       ax, WORD PTR [ebx+edx*2]
```

Listing 9.3: Assembler optimized C line

This is actually doing exactly the same and it even using one instruction less. One instruction is not much, but if there are 12 instructions in the loop, two instructions less is 16.666% less. The only problem is how to explain this for the C compiler.

The C compiler goes through a line from left to right. It first operates the inner most operations and then the higher once, but it always begins to read line from the left. So the first operation it parses is the shift, and then it will notice the inner AND operation, which is executed before the shift. A rough estimate is that the compiler caches the value in edx for the shift operation and goes to work on the inner operation first. What would happen if we move the shift operation after the AND operation?

We can actually see that in the 32 bit table-driven algorithm the shift operation is after the AND operation. There really was no reason bigger than random event that those were swapped to the 16 bit table-driven algorithm. The new C code is following:

```
unsigned short
crcr_algorithm (const void *data, unsigned int len)
{
         const unsigned short *end;
         const unsigned short *ptr;
         const unsigned short *lut;
         unsigned short reg;

         end = (const unsigned short*)
                          (((unsigned int)data) + len);
         ptr = (const unsigned short*)data;
         lut = (const unsigned short*)(table);
         reg = 0x0000FFFF;

         while (ptr < end) {
             reg ^= *--end;
             reg = lut[reg & 0xFF] ^ (reg >> 8);
             reg = lut[reg & 0xFF] ^ (reg >> 8);
         }

         return reg;
}
```

Listing 9.4: The final C code after assembler checkout

After the operations were swapped the assembler output turned out to:

```
.L6:
         sub      ecx, 2
         xor      ax, WORD PTR [ecx]
         movzx    edx, al
         shr      ax, 8
         xor      ax, WORD PTR [ebx+edx*2]
         movzx    edx, al
         shr      ax, 8
         xor      ax, WORD PTR [ebx+edx*2]
         cmp      esi, ecx
  jb     .L6
```

Listing 9.5: Assembler output after the last C modifications

By swapping the operations we got two instructions less code, which is rather interesting. The compiler optimization is out of scope of this project, so we do not go any further in this. As a final thing, a random answer from Gcc community:

“Deciding which variables and temporaries should go in which registers is a very hard problem. Compilers generally don't do it very well (compared to skilled hand coding). But not doing that well is not a "bug". At most it is "room for improvement".”

7.8. Benchmark

The benchmarking between the current and the new optimized algorithm was done in a simple way to measure how long it took to run the algorithm. This period of time is different between different computers, so we do not show any time based values in the results. Instead we show the values as “relatively faster/slower”. The amount of data used was 220 bytes, which was run 192 times for each algorithm. In the real implementation we are handling 22 byte packages, but the benchmark results with a small package would be highly obfuscated by the rest of the benchmark implementation.

The ten fastest rounds from the total run times were taken to the results, because those were least affected by external variables. As an example of external variable could be a task swap by Linux scheduler. To minimize external variables the benchmark was run with priority -20, which is the most prioritized process in Linux.

C code is as good as the used compiler is, so the benchmark was compiled and run with different optimization flags. The O1 -flag is optimization without any optimizations which take lot of time to compile. The O2 –flag uses nearly all supported optimizations, but does not use the ones which may greatly increase the size of binary. The O3 –flag uses all the possible optimizations the compiler provides.

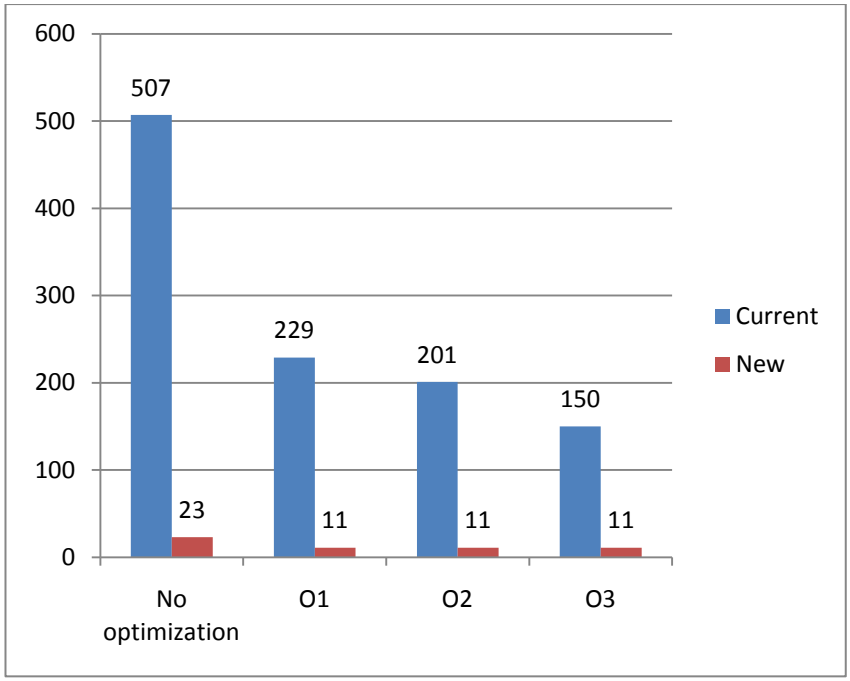


Image 10.1: Comparative performance between the algorithms, with the different compiler optimization flags.

8. CRC Class

This sprint implements all the knowledge acquired in the previous sprint, in the way that it can be easily used by other developers.

8.1. Requirements

Implement a class which provides an easy interface to use CRC algorithms. User has to only create the object and calculate CRC without extra function calls. Creation of lookup tables for CRC algorithms should be automatic, and created at the same time when creating the handle object. The implementation has to support the current 16 bit algorithm, but other algorithms might be added later. A CRC algorithm factory shall be kept on eye as a long term goal, even if it is not fully implemented in this project. In the first step the factory can create new CRC lookup tables from different poly and register base combinations.

If multiple objects are using the same arguments to create a lookup table, those shall automatically share the same lookup table in memory. Each lookup table is identified by the poly and register base combination. In the future identifier may also consist from the algorithm used.

8.2. Study

I have around five years of experience from C language, but the C++ language was new. In this phase it was mandatory to spent time to study how to design and implement in C++ language. The main material used was

countless random web pages. The time needed for this is reserved, because without good base knowledge it is impossible to create good, solid implementation.

8.3. Design

Five separated classes were built. The first one is Table class, to wrap a lookup table. The second class is RefCounter class, which is a simple reference counter class. The third one is ref_ptr class which is an auto pointer class. This is built to share a memory used by one lookup table. The fourth class is to implement AutoMap class. The main reasons are to map a lookup table to specific key, and to provide interface without memory management. The last class to implement is the handle class to use CRC algorithms. This class uses all other classes and also implements the algorithm factory.

8.3.1. Table

The lookup table is wrapped by Table class, to make lookup table creation and usage more comfortable. An array operator is provided to make the interface to be still like a normal array. In case this hits badly to the algorithm performance, a function returning a raw pointer to the table is provided. The Table supports only 16 bit lookup tables. In case if 32 or 64 bit lookup tables are needed, extra methods will be added by overloading the previous ones or adding with new names. The Table is not done as a template because it has very strict way of use.

8.3.2. RefCount

The only services provided from this class are increasing and decreasing the reference count, and return the current count as well. The functionality is not complicated but it is put in own class to make the implementation of user classes to be simple. Another reason is that it is highly possible that someone needs a thread safe and portable implementation in the future. When well separated from the other classes this kind of enhancement is trivial to implement.

8.3.3. ref_ptr

The sole purpose for the class is to share the memory used by the lookup tables. The reference counting is keeping track of how many different handles there are to this lookup table. When all the handles are gone, all the references are gone, and the lookup table is freed. This class uses the RefCount class for reference counting. Completely implemented auto pointers usually have every possible operator implemented to make it work with every possible child type. This is not necessary at the moment, and only the features currently needed are implemented.

The class is done as a template class, so that it can be easily used for other purposes too. If used for other purposes it is highly recommended to remove it from crc namespace, but no other modifications should be mandatory.

8.3.4. AutoMap

A map class is needed to keep track of built lookup tables. The class is more or less the same as any other map class. The difference is that the class is designed to work well with auto pointers. It is expected that an auto pointer is used and memory management is automatically handled when the map is needed no more. The ref_ptr class is used with AutoMap. The map template uses two types, first is the key type and the second is the auto pointer type. These gives space for possible enhancements in the future. The normal container functionality is implemented from the parts currently needed.

The class is done as a template class, so that it can be easily used for other purposes too. If used for other purposes it is highly recommended to remove it from crc namespace, but no other modifications should be mandatory. The ref_ptr class in most cases is good to keep in the same package.

8.3.5. Checker class

Two methods are provided, one to calculate CRC values and another to check if the result is the value we have already. The calculate function returns the calculated CRC from data and length of the data. The check function takes also the data and length as parameters, but also the CRC value to check against. The check function returns a Boolean value one if the CRC matches, else zero.

8.4. Implementation

The idea of this chapter is to introduce the main points of the implementation, and tell why things were done as those were done. Most of the implementation is straight forward, so only uncommon or big main details have been gathered out. The chapter is divided under five separated subtitles, one for each class. The minimum has been implemented in the header, to keep the interface well readable. The source and headers can be found from the appendix.

8.4.1. RefCount

The Implementation is almost like yelling atomic implementation, which shall be done when taken in multi thread environment. For debugging reasons the destructor has been made to abort if the count is not zero.

8.4.2. ref_ptr

The class is implemented as a template class. Only the currently needed operators have been implemented. Equal -operator allows comparing against zero to see if there is a child in.

8.4.3. AutoMap

The class is implemented as a template class. It uses std list as an internal container. A separated structure is used to save the key and value pair. These structures are then places in a std list to create the map feature. When the map is destroyed the list and the keys are destroyed as well. The memory management is left for the types passed when the template class was created. Even that template classes should be usable by any type, here it is forced to use types which handle the memory management by them self. This way the end user does not have to write destruction function to release memory allocated by the map and its children. This removes some versatility but offers simplicity.

The std list is not the most efficient container type for this, but with the data amounts we use, it is. As a future improvement the AutoMap could implement same interfaces as std containers.

8.4.4. Table

The class offers creation of lookup tables for one type of algorithms. Currently it is not possible to use external algorithms. As an example, an algorithm may have to take care of the byte order. Only 16 bit algorithms are possible, but internally the current algorithm is a template function to create the lookup tables. The reason is that it is possible to create lookup tables for other bit widths if the same method can be used. This is not in use currently because it is not needed, and secondly it is not certain if the method would be ok for other bit widths. In the future the class could offer of this method and/or use of external algorithms.

8.4.5. Checker

This class is the main class which uses all the services provided by the other classes. This is the handle class to be created for CRC calculations. It uses internally the factory and when the factory gets more features the API will be changed. The class implements the default 16 bit algorithm to be used at CERN.

Sharing lookup tables is done by using ref_ptr auto pointer with every lookup table. The auto pointers are then put in the AutoMap object, where those can be found by using the poly/register base key. This key is declared as a CrcKey structure. The map object holds one reference count in ref_ptr auto pointers so that none of the lookup tables will be freed before the AutoMap is destroyed. This way the handle objects can be created and freed lightly, knowing that the lookup tables are created only once. Each handle object contains ref_ptr to the Table object, and when the handle is destroyed the ref_ptr is destroyed, and the reference to Table is freed. The map object is static and automatically freed when the application ends.

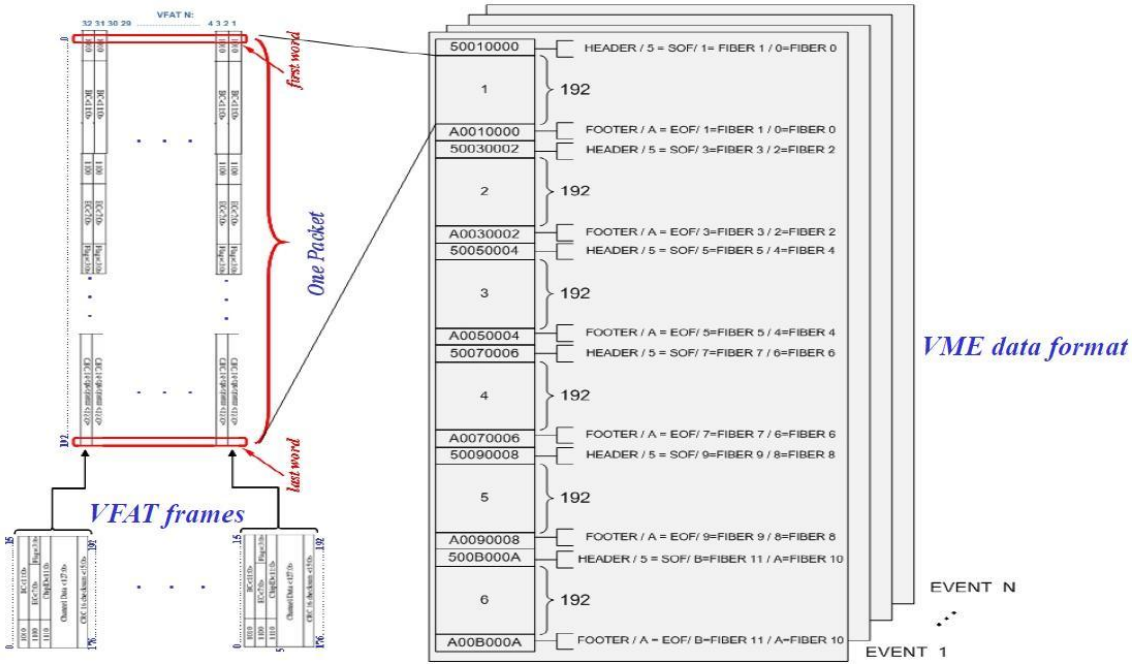
The map is providing multiple different algorithms or poly to be used in the same application. This is actually rather rare need and it could be enough to use only std list instead of a map. This would mean that there could be only one poly and register base pair which would be defined at the initialization phase. The reason to still use map, is that it does not need lot of extra work and it gives more options for the future enhancements.

9. The emulator class

In this chapter we go through the design and implementation of the emulator class. Shortly, the emulator is creating similar packets as there were something real connected to the network interface of a PC. This means that we have to write data in correct layout and also take care that the data looks valid, from parts it is needed. The chapter explains the hardware behind the frames, and why things are implemented in this specific way.

9.1. Data acquisition frames

The names for the frames come from the hardware places where those are constructed. The data acquisition frames are VFAT frame, OptoRx frame and TOTFED frame. VFAT frames are the smallest ones (24 bytes), and one OptoRx frame contains 32 VFAT frames. TOTFED is the biggest frame and contains maximum of six OptoRx frames.

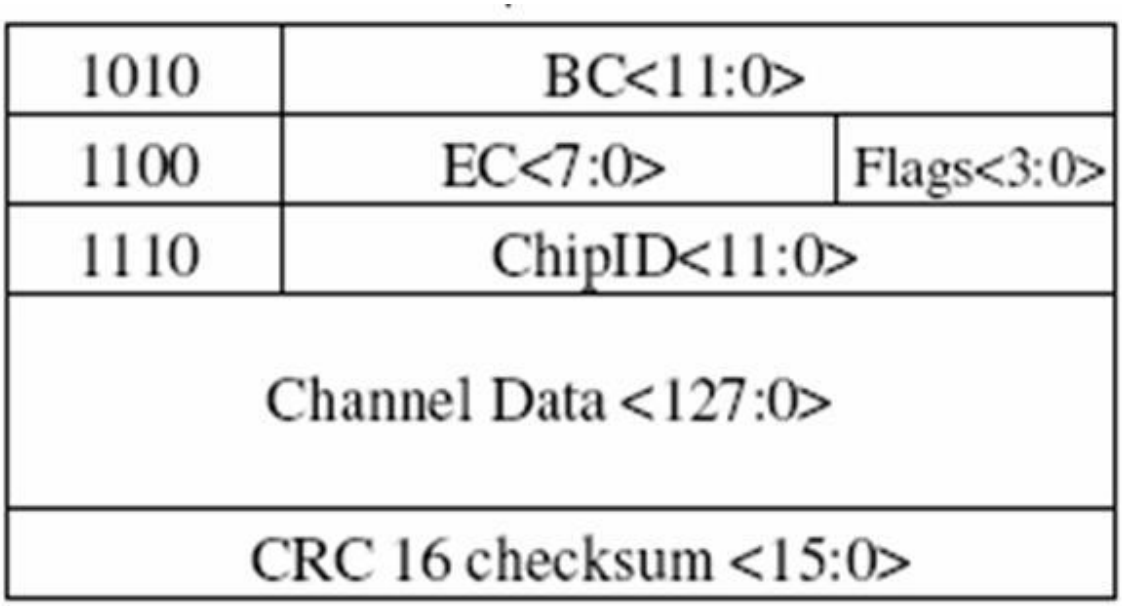


Picture 9.1: Data acquisition frames structure.

9.1.1. VFAT frame

The VFAT frame consists six bytes of header information, the data and 16 bit CRC checksum. The data is the data read from the sensors and the CRC is the CRC checksum from the whole VFAT packet. The only thing needing more explanation is the header.

The header is divided in three, two byte fields. Each field has a constant number identifying that specific header field. The fields are Bunch Crossing (BC) identifier, Event Counter (EC) identifier including Flags, and Chip ID identifier. The following picture shows the details.



Picture 9.1.1: The VFAT frame

The BC number is 12 bit counter which increments on every VFAT clock cycle. The EC number is 8 bit counter which increments on every level one trigger signal. The flags are identifying the state in the VFAT chip. The chip ID identifies the chips from each other.

9.1.2. OptoRx frame

Each OptoRx packet contains data from two fibers. One fiber transfers 16 VFATs, so two fibers transfer 32VFATs. OptoRx receives data from VFATs parallel, so that the first bits of VFAT frames arrives first, then the second bits, etc. The data is also written directly in memory, so that the first 32 bits read from the memory, are the first bits from all 32 VFAT frames. We could say that the hardware serializes the data. The data in the OptoRx frame is in serialized form. The picture 9.1, shows one of these packets and how the VFAT frames are in the OptoRx frame.

9.1.3. TOT FED frame

The data from OptoRx is written to 12 separated buffers by the main FPGA on the TOTFED board, one buffer per each fiber. At the same time the header and footer is written. After this the packet is sent to a PC. The emulator is creating these same packets. In the picture 9.1 the TOTFED frame is the biggest one, the one with grayed background.

9.2. Software implementation

In this chapter we go through the classes and tools created to implement the emulator. The implementation was divided in multiple independent units. The implementation contains a tool to turn horizontal bits to vertical bit order. In this case the OptoRx emulator will use it to turn the bits of VFAT frames. The OptoRx emulator creates the data directly received by the

optorx. The VFAT has two classes. The first one is built to manage the frame and another one to generate realistic VFAT packets.

9.2.1. Horizontal to vertical

This tool was not implemented as a class, but instead implementation contains direct function calls. There are three functions, one top of each other. The bitcopy function copies one bit value from place A to B. This function is then used in the hor2ver_char implementation, which copies 8bytes at once. The hor2ver_char is then used in the hor2ver_area function. The hor2ver function copies a buffer from place A to B.

9.2.2. VFAT frame

The VFAT frame is full of small bit values, which are shown in the order in the following listing.

```
unsigned int BCCode : 4;
unsigned int bcnumber : 12;
unsigned int EventCode : 4;
unsigned int eventnumber : 8;
unsigned int flags : 4;
unsigned int ChipID : 4;
unsigned int chipid : 12;
char channeldata[16];
unsigned int crc : 16;
```

The application binary interface (ABI) of the bit fields is compiler specific. This means that the compiler can move the values in memory as it likes. The gnu compilers do not offer option to set this structure to be as it is, and ask compiler to skip moving of the bits. This is the reason why all the bit field operations have to be handled manually. The bit field syntax is nice, so it is rather sad that this kind of feature is not implemented. If the compiler would implement this kind of feature, there would still be portability problems.

The name of the class is HFrame, it is actually a bit misleading. The reason for this is that the current class is in horizontal bit order. This is leaving space if someone wants to write in the future a class also to manage vertical frames, like VFrame. Then both classes can be derived from dummy VFATFrame class, just to define and document. The next developer should still remember to not use virtual functions, because this would add an invisible pointer to a virtual table. This would simply break the ABI of the structure, or force the developer to write it so that the framedata locates always in the heap, and never in the stack.

In the implementation is get and set function pairs for each variable. This approach was taken, because it probably produces the nicest implementation.

9.2.3. VFAT generator

The VFAT generator uses the VFAT frame to generate packages which do look like real ones, so this is also an emulator in that sense. The generator also provides signals like in the VFAT chips. At the moment only the clear signal is implemented, but it is implemented so that new signals are easy to add. At the moment the signals are not needed in the emulator, and the only reason to write this was to create a base for the future.

9.2.4. OptoRx emulator

There is not OptoRx frame implementation, mainly because all the data is in serialized form. The packet is neither edited when it has been built to this state, so spending time for this is not urgent. Instead, an emulator is built to create these packages. The OptoRx frame contains all the data in serialized form

The hor2ver function is not highly optimized, but before really optimizing, the function should be benchmarked. The implementation is rather simple and clear, so there is no point to make it more complicated before benchmarking. Hopefully this is fast enough, but in case it is not, I would begin

to look optimization options from here. At the moment we copy all the bits, bit by bit every time. It would be possible to preset the constant values of VFAT frame to the memory. In this way 48 bit sized copies could be skipped.

9.2.5. TOTFED frame

The TOT FED frame is simply defining the structure of the frame. One frame consists from header, footer, subframe and frame. The frame contains multiple similar shells, which are called as subframes. Each subframe contains header, footer and the data.

9.2.6. The emulator

The emulator itself is a simple class using all other classes to do its job. It still fills the header and footer in a proper way. The rest of the implementation is following the same guide lines as the other classes.

References

1. http://lhc.web.cern.ch/lhc/
2. http://cms.web.cern.ch/cms/index.html
3. http://totem.web.cern.ch/Totem/
4. http://www.inst.bnl.gov/programs/gasnobledet/hepnp/csc.shtml
5. http://ts-dep-dem.web.cern.ch/ts-dep-dem/products/gem/
6. http://totem.web.cern.ch/Totem/work_dir/electronics/pdf%20files/VFAT2SpecV5.pdf
7. http://accelconf.web.cern.ch/AccelConf/e06/PAPERS/MOPLS013.PDF
8. http://www.scrumalliance.org/
9. http://trac.edgewall.org
10. http://www.agile42.com/cms/pages/agilo
11. http://www.gnu.org/software/gdb
12. http://valgrind.org
13. http://www.gnu.org/software/make/
14. http://projects.gnome.org/gedit/
15. http://www.vim.org
16. http://indico.cern.ch/getFile.py/access?contribId=29&sessionId=6&resId=0&materialId=slides&confId=54346
17. http://cdsweb.cern.ch/record/1069713/files/cer-002725469.pdf

Appendix

A: The project code

A: The project code

```
/** \file automap.hh
 *  \brief More or less like std::map, but is used with auto pointers.
 *         Memory management is left for the auto map and pointers.
 *
 *  \author Risto Kivilahti <kivilaht@cern.ch> */

#ifndef __AUTO_MAP_HH__
#define __AUTO_MAP_HH__

#include <list>
#include <cstring>
#include "ref_ptr.hh"


namespace crc
{


template <typename Key, typename AutoPtr>
struct KeyValuePair
{
           Key key;
           AutoPtr value;
};


template <typename Key, typename AutoPtr>
class AutoMap
{
public:
           ~AutoMap () throw ();
           void add (Key &key, AutoPtr &value) throw ();
           void remove (Key &key) throw ();
           AutoPtr* find (Key &key) const throw ();

private:
           std::list<KeyValuePair<Key,AutoPtr>* > m_items;
};

template <typename Key, typename AutoPtr>
AutoMap<Key,AutoPtr>::~AutoMap () throw ()
{
           typename std::list<KeyValuePair<Key,AutoPtr>* >::iterator
iter;

           for (iter=m_items.begin(); iter!=m_items.end(); iter++)
           {
                      delete *iter;
           }
}
```

```
/** \brief Adds a new item in the AutoMap.
 *  \param key Key to identify the data.
 *  This has to be unique through the table.
 *  \param value The value, the information saved.
 *  \todo If the key already exists in the AutoMap, replace it. */
template <typename Key, typename AutoPtr>
void
AutoMap<Key,AutoPtr>::add (Key &key, AutoPtr &value) throw ()
{
           KeyValuePair<Key,AutoPtr> *kvp;
           kvp = new KeyValuePair<Key,AutoPtr>;

           kvp->key = key;
           kvp->value = value;

           m_items.push_front (kvp);
}

/** \brief Remove an item identified by the key from the AutoMap
 *  \param key The key to identify the value to be removed.
 *         If there is no data for the key, does nothing. */
template <typename Key, typename AutoPtr>
void
AutoMap<Key,AutoPtr>::remove (Key &key) throw ()
{
           void *vkey = reinterpret_cast<void*> (&key);
           typename std::list<KeyValuePair<Key,AutoPtr>* >::iterator
iter;

           for (iter=m_items.begin(); iter!=m_items.end(); iter++)
           {
                      void *tmp = reinterpret_cast<void*> (&(*iter)-
>key);
                      if (memcmp (tmp, vkey, sizeof (Key)) == 0) {
                                 m_items.erase (iter);
                      }
           }
}
```

```
/** \brief Find an item identified by the key from the AutoMap.
 *  \param key The key to identify the value to lookup
 *  \retun Pointer to the found value, in AutoPtr Pointer.
 *
 *  \todo API to return pointer to auto pointer is a bit akward,
 *  check is there something todo for this ? */
template <typename Key, typename AutoPtr>
AutoPtr*
AutoMap<Key,AutoPtr>::find (Key &key) const throw ()
{
           void *vkey = reinterpret_cast<void*> (&key);
           typename std::list<KeyValuePair<Key,AutoPtr>*
>::const_iterator iter;

           for (iter=m_items.begin(); iter!=m_items.end(); iter++)
           {
                      void *tmp = reinterpret_cast<void*> (&(*iter)-
>key);
                      if (memcmp (vkey, tmp, sizeof (Key)) == 0) {
                                 return &(*iter)->value;
                      }
           }

           return 0;
}


} /* namespace crc */

#endif /* __REF_MAP_HH__ */
```

```
/** \file crcchecker.hh
 *  \brief CRChecksum checker.
 *  Calculates CRChecksum with table driven algorithm.
 *  Poly and the register base can be defined.
 *  New tables are created and memory managed by the Checker.
 *
 *  \author Risto Kivilahti <kivilaht@cern.ch> */

#ifndef __CRC_CHECKER_HH__
#define __CRC_CHECKER_HH__


namespace crc
{

const unsigned short g_default_16bit_poly = 0x8408;

class Checker
{
public:
           virtual ~Checker () {}

           virtual unsigned short calculate (const void *data,
unsigned int len) = 0;
           virtual bool check (const void *data, unsigned int len,
unsigned short crc) = 0;
};


Checker *newChecker (unsigned short poly, unsigned short regbase);


} // nampespace crc

#endif /* __CRC_CHECKER_HH__ */
```

```
/** \file crcchecker.cpp
 *  \brief CRChecksum checker.
 *  Calculates CRChecksum with table driven algorithm.
 *  Poly and the register base can be defined.
 *  New tables are created and memory managed by the Checker.
 *
 *  \author Risto Kivilahti <kivilaht@cern.ch>
 *
 *  \todo More table driven CRC features ?
 *  Like own defined algorithms and different table creation
 *  algorithms.  Reversed crc ?  etc. etc. */

#include <map>
#include "crcchecker.hh"
#include "automap.hh"
#include "crctable.hh"
#include "ref_ptr.hh"


namespace crc
{

/* This is the key used to map the crctables in the AutoMap */
struct CrcKey
{
          unsigned short poly;
          unsigned short base;
};


static AutoMap <CrcKey,ref_ptr<Table> > g_algorithms;


class CheckerImpl : public Checker
{
public:
          CheckerImpl (unsigned short poly, unsigned short base);
          unsigned short calculate (const void *data, unsigned int
len);
          bool check (const void *data, unsigned int len, unsigned
short crc);

private:
          unsigned short algorithm_16bit (const void *data, unsigned
int len);
          void fetchTable (const unsigned short poly, const unsigned
short base);
          ref_ptr<Table> m_table;
          unsigned short m_base;
};
```

```
void
CheckerImpl::fetchTable (const unsigned short poly, const unsigned
short base)
{
          ref_ptr <Table> table;
          CrcKey key = { poly, base };

          table = *g_algorithms.find (key);
          if (table == 0)
          {
                    table = newTable (poly, base);
                    g_algorithms.add (key, table);
          }

          m_base = base;
          m_table = table;
}


CheckerImpl::CheckerImpl (unsigned short poly, unsigned short base)
{
          fetchTable (poly, base);
}


/** \brief Calculates CRChecksum from the data passed.
 *  \param data The data to calculate CRC from.
 *  \param len The lenght of the data.
 *  \return Calculated CRC */
unsigned short
CheckerImpl::calculate (const void *data, unsigned int len)
{
          return algorithm_16bit (data, len);
}

/** \brief Calculates CRChecksum from the data passed and compares it
to the crc
 *  \param data The data to calculate CRC from.
 *  \param len The lenght of the data.
 *  \param crc The CRC to compare against.
 *  \return 1 if the CRChecksums are the same, else 0. */
bool
CheckerImpl::check (const void *data, unsigned int len, unsigned short
crc)
{
          return (algorithm_16bit (data, len) == crc);
}


Checker *
newChecker (unsigned short poly, unsigned short regbase)
{
          if (poly == 0)
          {
                    poly = g_default_16bit_poly;
          }

          CheckerImpl *impl = new CheckerImpl (poly, regbase);
          return impl;
}
```

```
unsigned short
CheckerImpl::algorithm_16bit (const void *data, unsigned int len)
{
            const unsigned short *end;
            const unsigned short *ptr;
            const unsigned short *lut;
            unsigned short reg;
            lut = m_table->get ();

            end = (const unsigned short*)(((unsigned int)data) + len);
            ptr = (const unsigned short*)data;
            reg = m_base;

            while (ptr < end)
            {
                        reg ^= *--end;
                        reg = lut[reg & 0xFF] ^ (reg >> 8);
                        reg = lut[reg & 0xFF] ^ (reg >> 8);
            }

            return reg;
}


} // namespace crc
```

```
/** \file crctable.hh
 *  \brief CRC Table is the table for table driven algorithm.
 *
 *  \author Risto Kivilahti <kivilaht@cern.ch>
 *
 *  \todo Future support for 32 and 64 bit CRC table */

#ifndef __CRC_TABLE_HH__
#define __CRC_TABLE_HH__


namespace crc {

class Table
{
public:
           virtual ~Table () {}

           virtual unsigned short operator[] (unsigned short) = 0;
           virtual const unsigned short *get (void) const throw () =
0;
};

Table *newTable (const unsigned short poly, const unsigned short
regbase);

};

#endif /* __CRC_TABLE_HH__ */
```

```
/** \file crctable.cpp
 *  \brief CRC Table is the table for table driven algorithm.
 *
 *  \author Risto Kivilahti <kivilaht@cern.ch> */

#include "crctable.hh"
#include "stdlib.h"


namespace crc {

class TableImpl : public Table
{
public:
           TableImpl (void *table);
           ~TableImpl ();

           const unsigned short *get (void) const throw ();

           unsigned short operator[] (unsigned short id);
           unsigned int operator[] (unsigned int id);

           unsigned int* operator= (unsigned int* p);
           unsigned short* operator= (unsigned short* p);

private:
           void *m_table;
};


TableImpl::TableImpl (void *table)
{
           m_table = table;
}

TableImpl::~TableImpl ()
{
           free (m_table);
}

/** \brief Return a pointer to the raw data of the table.
 *  This is unsigned short, so use with 16 bit algorithm.
 *  You can use either operator[], or this function to get the raw
data.
 *  When used in algorithm loop, the operator[] seems to be more or
less 3 times
 *  slower than the raw data.
 *  \return The pointer to the raw data. */
const unsigned short *
TableImpl::get (void) const throw ()
{
           return (unsigned short*)m_table;
}

unsigned short
TableImpl::operator[] (unsigned short id)
{
           return ((unsigned short*)m_table)[id];
}
```

```
unsigned int
TableImpl::operator[] (unsigned int id)
{
           return ((unsigned int*)m_table)[id];
}

unsigned int*
TableImpl::operator= (unsigned int* p)
{
           m_table = p; return p;
}

unsigned short*
TableImpl::operator= (unsigned short* p)
{
           m_table = p; return p;
}


template <class T>
static void
calculate_lut (T *table, T poly, T regbase)
{
           int i, j;
           T val = regbase;

           for (i=0; i<256; i++) {
                      val = i;

                      for (j=0; j<8; j++) {
                                 if (val & 0x1) {
                                            val = (val >> 1) ^ poly;

                                 } else {
                                            val = val >> 1;
                                 }
                      }

                      table[i] = val;
           }
}

Table *
newTable (const unsigned short poly, const unsigned short regbase)
{
           unsigned short *table = (unsigned short*)malloc (256 *
sizeof (unsigned short));
           TableImpl *timpl = new TableImpl (table);

           calculate_lut<unsigned short> (table, poly, regbase);

           return dynamic_cast<Table*>(timpl);
}

} /* namespace crc */
```

```
/** \file hor2ver.hh
 *  \brief Converts horizontal bits to vertical bits
 *
 *  Horizontal:
 *  1111
 *  0000
 *  1111
 *  0000
 *
 *
 *  Vertical:
 *  1010
 *  1010
 *  1010
 *  1010
 */

#ifndef __HORIZONTAL_TO_VERTICAL_HH__
#define __HORIZONTAL_TO_VERTICAL_HH__


void hor2ver_area (const unsigned char *src, unsigned int amount,
           unsigned char *dest, unsigned char destbit, unsigned int
destwidth);

inline void bitcopy (const unsigned char *from, unsigned char frombit,
                     unsigned char *to, unsigned char tobit);

void hor2ver_char (const unsigned char *from, unsigned char *to,
                     unsigned char tobit, unsigned int towidth);


#endif /* __HORIZONTAL_TO_VERTICAL_HH__ */
```

```
/** \file hor2ver.cpp */

#include "hor2ver.hh"


void
hor2ver_area (const unsigned char *src, unsigned int amount, unsigned
char *dest,
                      unsigned char destbit, unsigned int destwidth)
{
           for (unsigned int k=0; k<amount; k++) {
                      hor2ver_char (src, dest, destbit, destwidth);
                      src++;
                      dest += destwidth*8;
           }
}

void
hor2ver_char (const unsigned char *from, unsigned char *to, unsigned
char tobit,
                      unsigned int towidth)
{
           for (unsigned char i=0x80; i!=0x00; i>>=1) {
                      bitcopy (from, i, to, tobit);
                      to+=towidth;
           }
}


inline void
bitcopy (const unsigned char *from, unsigned char frombit, unsigned
char *to,
           unsigned char tobit)
{
           if (*from & frombit) {
                      *to |= tobit;
           }
}
```

```
/** \file optorxgenerator.hh */

#ifndef __OPTORX_GENERATOR_HH__
#define __OPTORX_GENERATOR_HH__


namespace totfedemu
{


class OptoRxGenerator
{
public:
           virtual ~OptoRxGenerator () {};
           virtual unsigned char *generateTo (unsigned char *dataptr)
= 0;
};


OptoRxGenerator *newOptoRxGenerator (void);


} /* namespace optorx */

#endif /* __OPTORX_GENERATOR_HH__ */
```

```
/** \file optorxemulator.cpp */

#include "optorxemulator.hh"
#include "vfatgenerator.hh"
#include "hor2ver.hh"
#include <string.h>

namespace totfedemu {


class OptoRxGeneratorImpl : public OptoRxGenerator
{
private:
           VFATGenerator *m_vfatgen;
           const vfat::HFrame *m_frame;

public:
           OptoRxGeneratorImpl ();
           ~OptoRxGeneratorImpl ();
           const unsigned char *generate (void);
           unsigned char *generateTo (unsigned char *dataptr);
};


OptoRxGeneratorImpl::OptoRxGeneratorImpl ()
{
           m_vfatgen = NewVFATGenerator (32);
}

OptoRxGeneratorImpl::~OptoRxGeneratorImpl ()
{
           delete m_vfatgen;
}


unsigned char *
OptoRxGeneratorImpl::generateTo (unsigned char *dataptr)
{
           memset (dataptr, 0, sizeof(dataptr));
           /* Defines the vertical byte to write */
           for (int k=0; k<4; k++) {
                      /* Defines the bit in the byte to write */
                      for (int i=0x80; i!=0x00; i>>=1) {
                                 m_frame = m_vfatgen->generate_next ();
                                 hor2ver_area ((const unsigned
char*)m_frame,

           sizeof(vfat::HFrame), dataptr+k, i, 4);
                      }
           }

           /* Reading the NULL out. */
           m_vfatgen->generate_next ();

           return dataptr;
}
```

```
OptoRxGenerator *
newOptoRxGenerator (void)
{
            return new OptoRxGeneratorImpl;
}


} /* namespace totfedemu */
```

```
/** \file ref_ptr.hh
 *  \brief Reference calculated pointer.
 *  Every copy of ref_ptr shares a reference count with
 *  the original ref_ptr.
 *  When copy is created, reference count is increased, and when copy
 *  is deleted, reference count is decreased.
 *  When the count reaches zero, the object will be destroyed.
 *
 *  There is no real API to use the object, but instead it should be
 *  used like the auto_ptr.  Check auto_ptr manual for more.
 *
 *
 *  \author Risto Kivilahti <kivilaht@cern.ch> */

#ifndef __REF_PTR_HH__
#define __REF_PTR_HH__

#include "refcount.hh"
#include <assert.h>


namespace crc
{

template <typename T>
class ref_ptr
{
public:
           ref_ptr () throw ();
           ref_ptr (T *ptr) throw ();
           ref_ptr (const ref_ptr<T> &ptr) throw ();
           virtual ~ref_ptr () throw ();

           ref_ptr<T>& operator= (T *ptr) throw ();
           ref_ptr<T>& operator= (const ref_ptr<T> &ptr) throw ();
           T& operator* () const throw ();
           T* operator-> () const throw ();

           bool operator== (const ref_ptr<T> &ptr) const throw ();

private:
           void unBind (void) const throw ();

           mutable T *m_object;
           mutable RefCount *m_refcount;
};


template <typename T>
ref_ptr<T>::ref_ptr () throw ()
{
           m_object = 0;
           m_refcount = 0;
}
```

```
template <typename T>
ref_ptr<T>::ref_ptr (T *ptr) throw ()
{
            m_object = 0;
            m_refcount = 0;

            if (ptr == 0) {
                        return;
            }

            m_object = ptr;
            m_refcount = new RefCount ();
            assert (m_refcount != 0);
            m_refcount->inc ();
}


template <typename T>
ref_ptr<T>::ref_ptr (const ref_ptr &ptr) throw ()
{
            m_object = 0;
            m_refcount = 0;

            if (&ptr == 0) {
                        return;
            }

            if (ptr.m_object != 0) {
                        assert (ptr.m_refcount != 0);
                        ptr.m_refcount->inc ();
                        unBind ();
                        m_object = ptr.m_object;
                        m_refcount = ptr.m_refcount;
            }
}


template <typename T>
ref_ptr<T>::~ref_ptr () throw ()
{
            unBind ();
}


template <typename T>
ref_ptr<T>&
ref_ptr<T>::operator= (T *ptr) throw ()
{
            unBind ();
            if (ptr == 0) {
                        return *this;
            }

            m_object = ptr;
            m_refcount = new RefCount ();
            assert (m_refcount != 0);
            m_refcount->inc ();
            return *this;
}
```

```
template <typename T>
ref_ptr<T>&
ref_ptr<T>::operator= (const ref_ptr<T> &ptr) throw ()
{
            unBind ();

            if (&ptr == 0) {
                        return *this;
            }

            if (ptr.m_object == 0) {
                        return *this;
            }

            assert (ptr.m_refcount != 0);

            if (ptr.m_refcount->inc () == 0) {
                        return *this;
            }

            m_object = ptr.m_object;
            m_refcount = ptr.m_refcount;
            return *this;
}


template <typename T>
T&
ref_ptr<T>::operator* () const throw ()
{
            return *m_object;
}


template <typename T>
T*
ref_ptr<T>::operator-> () const throw ()
{
            return m_object;
}


template <typename T>
bool
ref_ptr<T>::operator== (const ref_ptr<T> &ptr) const throw ()
{
            if (&ptr == 0) {
                        return (m_object == 0);
            }

            return (m_object == ptr.m_object);
}
```

```
template <typename T>
void
ref_ptr<T>::unBind (void) const throw ()
{
            if (m_object != 0) {
                        assert (m_refcount != 0);
                        if (m_refcount->dec () == 0) {
                                    delete m_object;
                                    delete m_refcount;
                        }

                        m_object = 0;
                        m_refcount = 0;
            }
}

} /* namespace crc */

#endif /* __REF_PTR_HH__ */
```

```
/** \file refcount.hh
 *  \brief Reference counter.
 *  Simple reference counter class.
 *
 *  \author Risto Kivilahti <kivilaht@cern.ch>
 *
 *  \todo Make thread safe, atomic. */

#ifndef __REF_COUNT_HH__
#define __REF_COUNT_HH__


namespace crc
{


class RefCount
{
public:
           RefCount (void) throw ();
           ~RefCount (void) throw ();

           unsigned int inc (void) throw ();
           unsigned int dec (void) throw ();

           unsigned int get (void) const throw ();

private:
           unsigned int m_count;
};


} /* namespace crc */

#endif /* __REF_COUNT_HH__ */
```

```
/** \file refcount.cpp
 *  \brief Reference counter.
 *
 *  \author Risto Kivilahti <kivilaht@cern.ch>
 *
 *  \todo Make thread safe */


#include "refcount.hh"
#include <assert.h>

namespace crc
{


RefCount::RefCount () throw ()
{
           m_count = 0;
}

RefCount::~RefCount () throw ()
{
           assert (m_count == 0);
}

/** \brief Return the current reference count.
 *  \return The current reference count. */
unsigned int
RefCount::get (void) const throw ()
{
           return m_count;
}

/** \brief Increase reference count by one.
 *  \todo There is no protection for overflow,
 *  but can someone really need ?
 *  \return The current reference count. */
unsigned int
RefCount::inc (void) throw ()
{
           return ++m_count;
}

/** \brief Decrease the reference count by one.  If count is zero,
does nothing.
 *  \return The current reference count. */
unsigned int
RefCount::dec (void) throw ()
{
           if (m_count == 0) {
                      return 0;
           }

           return --m_count;
}


} /* namespace crc */
```

```
/**  \file totfedemulator.hh */

#ifndef __TOTFED_EMULATOR_HH__
#define __TOTFED_EMULATOR_HH__

#include "totfedframe.hh"

namespace totfedemu
{


class FrameEmulator
{
public:
           virtual ~FrameEmulator () {}

           /* Generates a TOT FED frame.  Running numbers in VFAT
            * frame are updated as should, like event number.  A new
            * buffer is always allocated, so the user is responsible
            * to free it. */
           virtual Frame *generate (void) = 0;
};


FrameEmulator* newFrameEmulator (unsigned int firstfiber);


} /* namespace totfedemu */

#endif /* __TOTFED_EMULATOR_HH__ */
```

```
/** \file totfedemulator.cpp */

#include "totfedemulator.hh"
#include "optorxemulator.hh"
#include <string.h>


namespace totfedemu {


class FrameEmulatorImpl : public FrameEmulator
{
public:
           FrameEmulatorImpl (unsigned int firstfiber);
           ~FrameEmulatorImpl (void);

           Frame *generate (void);

private:
           unsigned int m_firstfiber;
           OptoRxGenerator *m_optogen[6];
           void generateHeader (Header *header, unsigned int fiber);
           void generateFooter (Footer *footer, unsigned int fiber);
           void generateSubFrame (SubFrame *subframe, unsigned int
fiber);
};


FrameEmulatorImpl::FrameEmulatorImpl (unsigned int firstfiber)
{
           m_firstfiber = firstfiber;
           for (int i=0; i<6; i++) {
                      m_optogen[i] = newOptoRxGenerator ();
           }
}

void
FrameEmulatorImpl::generateHeader (Header *header, unsigned int fiber)
{
           header->SOFCode = 0x50; /* Const SOF Code */
           header->afiber = fiber;
           header->notused = 0x00; /* Field not used */
           header->bfiber = ++fiber;
}

void
FrameEmulatorImpl::generateFooter (Footer *footer, unsigned int fiber)
{
           footer->EOFCode = 0xA0; /* Const EOF Code */
           footer->afiber = fiber;
           footer->notused = 0x00; /* Field not used */
           footer->bfiber = ++fiber;
}
```

```
void
FrameEmulatorImpl::generateSubFrame (SubFrame *subframe, unsigned int
fiber)
{
           generateHeader (&subframe->header, fiber);
           generateFooter (&subframe->footer, fiber);

           /* Hack: fiber - m_firstfiber.  For each fiber pair there
            * is one OptoRxGenerator.  The ID of generator follows
            * linearly the fiber nbr.
            * So we use fiber nbr to select correct OptoRxGenerator.
            */
           m_optogen[fiber - m_firstfiber]->generateTo (
                                (unsigned char*)(subframe->data));
}

totfedemu::Frame *
FrameEmulatorImpl::generate (void)
{
           Frame *frame = new Frame;
           memset (frame, 0, sizeof(Frame));

           for (int i=0; i<6; i++) {
                      generateSubFrame (&(frame->subframe[i]),
                                m_firstfiber+i);
           }

           return frame;
}

FrameEmulatorImpl::~FrameEmulatorImpl (void)
{
           for (int i=0; i<6; i++) {
                      delete m_optogen[i];
           }
}

/* \brief Returns new FrameEmulator.
 * \param firstfiber The first fiber ID, the rest are sequential.
 * \return Newly allocated FrameEmulator */
FrameEmulator*
newFrameEmulator (unsigned int firstfiber)
{
           return new FrameEmulatorImpl (firstfiber);
}


} /* namespace daqemu */
```

```
/** \file totfedframe.hh */

#ifndef __TOTFED_FRAME_HH__
#define __TOTFED_FRAME_HH__

namespace totfedemu {


struct Header
{
           char SOFCode; // = 0x50 const
           char afiber;
           char notused; // = 0x00 const
           char bfiber;
};

struct Footer
{
           char EOFCode; // = 0xA0 const
           char afiber;
           char notused; // = 0x0 const
           char bfiber;
};

struct SubFrame
{
           Header header;
           unsigned int data [192];
           Footer footer;
};

struct Frame
{
           SubFrame subframe[6];
};


} /* namespace totfed */

#endif /* __TOTFED_FRAME_HH__ */
```

```
/** \file vfatframe.hh */

#ifndef __VFAT_FRAME_HH__
#define __VFAT_FRAME_HH__

namespace vfat
{


/*
           VFAT frame is like this structure.  Const in some fields
           means that the value of this field is set to const value.

           unsigned int BCCode : 4; // = 10 const;
           unsigned int bcnumber : 12;
           unsigned int EventCode : 4; // = 12 const;
           unsigned int eventnumber : 8;
           unsigned int flags : 4; // = 0;
           unsigned int ChipCode : 4; // = 14 const;
           unsigned int chipid : 12;
           char channeldata[16];
           unsigned int crc : 16;
*/

class HFrame
{
private:
           /* There are here to verify the data in the framedata, in
case needed */
           unsigned int getBCCode (void) const;
           unsigned int getEventCode (void) const;
           unsigned int getChipCode (void) const;

public:
           char framedata[24];
           HFrame ();

           void setBCNumber (unsigned int bcnbr);
           unsigned int getBCNumber (void) const;

           void setEventNumber (unsigned int eventnbr);
           unsigned int getEventNumber (void) const;

           void setFlags (unsigned int flagsval);
           unsigned int getFlags (void) const;

           void setChipID (unsigned int chipid);
           unsigned int getChipID (void) const;

           char *getDataPtr (void);

           void setCRC (unsigned int crc);
           unsigned int getCRC (void) const;

           void print (void) const;
};

} /* namespace vfat */

#endif /* __VFAT_FRAME_HH__ */
```

```
/** \file vfatframe.cpp
 *  \brief VFATFrame and handling of the bitfields. */

#include <stdio.h>
#include "vfatframe.hh"


namespace vfat
{

static const char BCCODE = 0xa0;
static const char EVENTCODE = 0xc0;
static const char DEFAULTFLAGS = 0x00;
static const char CHIPCODE = 0xe0;


HFrame::HFrame ()
{
           framedata[0] = BCCODE;
           framedata[2] = EVENTCODE;
           framedata[3] = DEFAULTFLAGS;
           framedata[4] = CHIPCODE;
}

unsigned int
HFrame::getBCCode (void) const
{
           return (unsigned int)(framedata[0] >> 4) & 0x0F;
}

unsigned int
HFrame::getEventCode (void) const
{
           return (unsigned int)(framedata[2] >> 4) & 0x0F;
}

unsigned int
HFrame::getChipCode (void) const
{
           return (unsigned int)(framedata[4] >> 4) & 0x0F;
}

void
HFrame::setBCNumber (unsigned int bcnbr)
{
           /* Write high bits */
           framedata[0] = (framedata[0] & 0xF0) | (char) ((bcnbr >> 8)
& 0x0F);

           /* Write low bits */
           framedata[1] = (char) (bcnbr & 0xFF);
}
```

```
unsigned int
HFrame::getBCNumber (void) const
{
           /* Read high bits */
           unsigned int ret = (unsigned int)framedata[0] & 0x0F;
           ret <<= 8;

           /* Read low bits */
           ret |= (unsigned int)framedata[1];

           return ret & 0x0FFF;
}

void
HFrame::setEventNumber (unsigned int eventnbr)
{
           /* Write high bits */
           framedata[2] = (framedata[2] & 0xF0) | (char)((eventnbr &
0xF0) >> 4);

           /* Write low bits */
           framedata[3] = (framedata[3] & 0x0F) | (char)((eventnbr &
0x0F) << 4);
}

unsigned int
HFrame::getEventNumber (void) const
{
           /* Write high bits */
           unsigned int ret = (unsigned int)((framedata[2] & 0x0F) <<
4);

           /* Write low bits */
           ret |= (unsigned int)((framedata[3] & 0xF0) >> 4);

           return ret & 0xFF;
}

void
HFrame::setFlags (unsigned int flagsval)
{
           framedata[3] = (framedata[3] & 0xF0) | (char)(flagsval &
0x0F);
}

unsigned int
HFrame::getFlags (void) const
{
           return (unsigned int)(framedata[3] & 0x0F);
}
```

```
void
HFrame::setChipID (unsigned int chipid)
{
           /* Write high bits */
           framedata[4] = (framedata[4] & 0xF0) | (char) ((chipid >>
8) & 0x0F);

           /* Write low bits */
           framedata[5] = (char) (chipid & 0xFF);
}

unsigned int
HFrame::getChipID (void) const
{
           /* Read high bits */
           unsigned int ret = (unsigned int)framedata[4] & 0x0F;
           ret <<= 8;

           /* Read low bits */
           ret |= (unsigned int)framedata[5];

           return ret & 0x0FFF;
}


char *
HFrame::getDataPtr (void)
{
           return &framedata[6];
}

void
HFrame::setCRC (unsigned int crc)
{
           framedata[22] = (char)((crc >> 8) & 0xFF);
           framedata[23] = (char)(crc & 0xFF);
}

unsigned int
HFrame::getCRC (void) const
{
           unsigned int ret = (((unsigned int)framedata[22]) << 8);
           ret |= (unsigned int)framedata[23];

           return ret & 0xFFFF;
}
```

```
void
HFrame::print (void) const
{
           printf ("BCCode:      %d\n"
                     "bcnumber:    %d\n"
                     "EventCode:   %d\n"
                     "eventnumber: %d\n"
                     "flags:       %d\n"
                     "ChipCode:    %d\n"
                     "chipid:      %d\n"
                     "channeldata: %.*s\n"
                     "crc:         %2X\n",
                     getBCCode (),
                     getBCNumber (),
                     getEventCode(),
                     getEventNumber (),
                     getFlags (),
                     getChipCode (),
                     getChipID (),
                     16,
                     &(this->framedata[6]),
                     getCRC ()
                     );
}

} /* namespace vfat */
```

```
/** \file vfatgenerator.hh */

#ifndef __VFAT_GENERATOR_HH__
#define __VFAT_GENERATOR_HH__

#include "vfatframe.hh"

namespace totfedemu
{


typedef void(*VFATDataEmulator)(char *channeldata, vfat::HFrame
*packet);


enum VFATSignal
{
           ClearSignal = 0,

           NumberOfSignals
};


class VFATGenerator
{
public:
           virtual ~VFATGenerator () {}
           /* Returns a VFATFrame with the next chipID, or 0 if none
            * is left. After 0 the loop begins again and you generate
            * the first one again. Loop is as long as vfatcount.
            * After every loop the event number is increased. */
           virtual const vfat::HFrame* generate_next (void) = 0;

           /* Emit VFATSignal for all the VFATs.  Currently only
            * ClearSignal is supported. */
           virtual void emit_signal (VFATSignal signal) = 0;
};


/* \brief Returns new VFATGenerator.
 * \param vfatcount The amount of VFATs in the generator.
 *        There is chipID for each of the VFATs. */
VFATGenerator* NewVFATGenerator (unsigned int vfatcount);


} /* namespace daqemu */

#endif /* __VFAT_GENERATOR_HH__ */
```

```
/** \file vfatgenerator.c
 *  \todo At the moment VFAT flags are not emulated in anyway and are
 *  always zero. */

#include <sys/time.h>
#include <stdlib.h>
#include <limits.h>
#include <string.h>
#include "vfatgenerator.hh"
#include "crcchecker.hh"

namespace totfedemu {


class VFATGeneratorImpl : public VFATGenerator
{
public:
           VFATGeneratorImpl (unsigned int count);
           ~VFATGeneratorImpl ();
           const vfat::HFrame* generate_next (void);
           void emit_signal (VFATSignal signal);

private:
           unsigned int m_vfatcount;
           unsigned int m_currentvfat;
           vfat::HFrame m_vfatgenbuf;
           VFATDataEmulator m_data_emu;
           crc::Checker *m_crcchecker;

           void channel_data_emulator(char *channeldata, const
vfat::HFrame *packet);
           void clear_signal (void);
};


VFATGeneratorImpl::VFATGeneratorImpl (unsigned int count)
{
           m_data_emu = 0;
           m_vfatcount = count;
           m_currentvfat = 0;
           m_crcchecker = crc::newChecker (0x8888, 0xFFFF);
}

VFATGeneratorImpl::~VFATGeneratorImpl ()
{
           delete m_crcchecker;
}
```

```
const vfat::HFrame*
VFATGeneratorImpl::generate_next (void)
{
           struct timeval tv;

           if (m_currentvfat > m_vfatcount) {
                      m_currentvfat = 0;
                      /* Event is changed only when all the VFATs are
                       * created. */
                      unsigned int eventnbr =
                                 m_vfatgenbuf.getEventNumber ();
                      eventnbr++;
                      m_vfatgenbuf.setEventNumber (eventnbr);
                      return NULL;
           }

           /* Set chipIP */
           m_vfatgenbuf.setChipID (m_currentvfat);

           /* Set BCN */
           gettimeofday (&tv, NULL);
           m_vfatgenbuf.setBCNumber (tv.tv_usec);

           /* Set channel data */
           channel_data_emulator (m_vfatgenbuf.getDataPtr (),
&m_vfatgenbuf);

           /* Set CRC */
           m_vfatgenbuf.setCRC (m_crcchecker->calculate
(&m_vfatgenbuf,

sizeof (vfat::HFrame) - 2));

           m_currentvfat++;
           return &m_vfatgenbuf;
}


void
VFATGeneratorImpl::clear_signal (void)
{
           m_currentvfat = 0;
           m_vfatgenbuf.setBCNumber (0);
           m_vfatgenbuf.setEventNumber (0);
}

void
VFATGeneratorImpl::emit_signal (VFATSignal signal)
{
           switch (signal)
           {
                      case ClearSignal:
                                 clear_signal ();
                                 break;

                      default:
                                 break;
           };
}
```

```
void
VFATGeneratorImpl::channel_data_emulator (char *channeldata, const
vfat::HFrame *packet)
{
           unsigned int *intdata = (unsigned int*)channeldata;

           srand (packet->getBCNumber ());

           /* int is assumed to be 32bit long. */
           for (int i=0; i<4; i++) {
                      /* Get some random data. */
                      intdata[i] = (unsigned int)(1000000.0 * (rand() /
3555555.0));
           }
}

VFATGenerator*
NewVFATGenerator (unsigned int vfatcount)
{
           VFATGenerator *vfatgen = new VFATGeneratorImpl (vfatcount);
           return vfatgen;
}

} /* namespace daqemu */
```